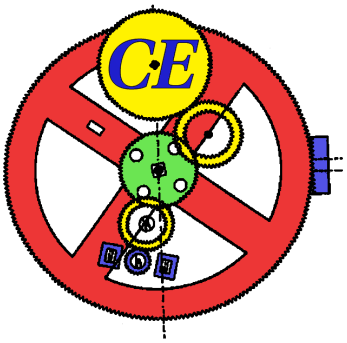

Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2011

MSc THESIS

Multi-input Embedded Real-time Software Defined Radio

Asif Jalaludeen

CE-MS-2011-08

This master thesis work is inspired by the practical aspects of the **Analytical Constant Modulus Algorithm (ACMA)** proposed by Alle-Jan van der Veen and Arogyaswami Paulraj. The ACMA deals with the beamforming problem associated with constant modulus co-channel signal interference in wireless communication. Co-channel interference occur when multiple signals are transmitted simultaneously at the same frequency from different sources. Beamforming is a technique applied in spatial signal processing to separate out individual signals using an antenna array. ACMA provides an efficient analytical approach to solve beamforming. It is a *blind beamforming* algorithm as it does not require any knowledge of the signals and the channels. The scope of this thesis work is to implement a low cost, low power, embedded Multi-input receiver application. The Multi-input receiver system shall handle partially and fully overlapping constant modulus signals from distinct sources. The signals are modulated using a generic modulation scheme. The Multi-input receiver shall separate out individual signals using the ACMA (blind beamforming) and demodulate each signal. The system shall be software defined such that the beamforming and demodulator are implemented in software on an embedded Digital Signal Processor (DSP) platform. The Multi-input receiver system shall be optimized for the DSP platform in order to achieve real-time performance in terms of speed and stay within the power budget of the system. The receiver shall work efficiently in the presence of interferences and noise such as thermal noise generated by the receiver.

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science

Multi-input Embedded Real-time Software Defined Radio

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

by

Asif Jalaludeen
born in Kollam, India

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Multi-input Embedded Real-time Software Defined Radio

by Asif Jalaludeen

Abstract

This master thesis work is inspired by the practical aspects of the **Analytical Constant Modulus Algorithm (ACMA)** proposed by Alle-Jan van der Veen and Arogyaswami Paulraj. The ACMA deals with the beamforming problem associated with constant modulus co-channel signal interference in wireless communication. Co-channel interference occur when multiple signals are transmitted simultaneously at the same frequency from different sources. Beamforming is a technique applied in spatial signal processing to separate out individual signals using an antenna array. ACMA provides an efficient analytical approach to solve beamforming. It is a *blind beamforming* algorithm as it does not require any knowledge of the signals and the channels. The scope of this thesis work is to implement a low cost, low power, embedded Multi-input receiver application. The Multi-input receiver system shall handle partially and fully overlapping constant modulus signals from distinct sources. The signals are modulated using a generic modulation scheme. The Multi-input receiver shall separate out individual signals using the ACMA (blind beamforming) and demodulate each signal. The system shall be software defined such that the beamforming and demodulator are implemented in software on an embedded Digital Signal Processor (DSP) platform. The Multi-input receiver system shall be optimized for the DSP platform in order to achieve real-time performance in terms of speed and stay within the power budget of the system. The receiver shall work efficiently in the presence of interferences and noise such as thermal noise generated by the receiver.

Laboratory : Computer Engineering
Codenumber : CE-MS-2011-08

Committee Members :

Advisor:	Georgi N. Gaydadjiev, CE, TU Delft
Chairperson:	Koen Bertels, CE, TU Delft
Member:	Hans-Gerhard Gross, ST, TU Delft

to humanity

Contents

List of Figures

List of Tables

1 Introduction

Multi-input receivers are used in wireless communication to improve communication performance when there are multiple radio channels transmitted simultaneously. This technology is based on multiple antennas to receive signals.

1.1 Background

Beamforming is a spatial signal processing technique applied in wireless communication to extract overlapping signals (co-channel interference) using an antenna array. Overlapping of signals occur when multiple signals are transmitted at the same frequency and at the same time from distinct sources. The objective is to separate out individual signals without interferences. The beamformer computes the proper weight vector w_i so that the individual signals can be derived from the linearly combed output of the antennas. This situation can be mathematically described as,

$$\mathbf{X = A\ S}$$

where X is the measured data at the antenna array, S is the original source signal, A is array response matrix. The weight matrix $W = A\dagger$ the pseudo-inverse of A.

Beamforming is called **blind beamforming** when the weigh vectors are calculated only from the measured data and without any knowledge of signals or channels. Constant Modulus Algorithms such as Godard and CMA are generally used for blind beamforming [23]. CMA have the drawback that they are iterative in nature and have limited performance.

This master thesis work is inspired by the 'Analytical Constant Modulus Algorithm' (ACMA) [23] proposed by Alle-Jan van der Veen and Arogyaswami Paulraj. The ACMA algorithm provides an efficient analytical solution to *blind beamforming*.

ACMA solution to the blind beamforming problem demonstrates several advantages over other Constant Modulus Algorithm (CMA) schemes. The conventional CMAs can detect only one signal from a set of overlapping signals, where ACMA can detect multiple signals with an upper limit on the number of antennas. A multi-stage (iterative) CMA can be used to detect more than one signal, but it comes with the price of processing. Moreover in practice ACMA demonstrates better performance to CMA in detecting signals. Another advantage is that ACMA requires only a moderate amount of input signal samples.

This thesis work involves implementing a low cost, low power, Multi-input receiver system. The input data packet consists of a binary bit stream (message), Cyclic Redundancy Check (CRC-16) and start & end flags (8 bit). The bit streams (messages) are of fixed length. CRC are added to this bit stream to detect accidental errors occurring during transmission. A start flag and end flag are added to this in order to detect a data packet. The complete message is then modulated using a constant modulus modulation scheme. The data packets containing distinct messages are transmitted from different sources at the same frequency and at the same time. This results in partial or full overlap interference of the signals.

The Multi-input receiver is a software defined radio which can successfully extract individual signals negating the signal interferences and noise. It has three basic building blocks; *antenna array, tuner module* and *digital signal processor*. The front-end is an antenna array and the analog tuner module. The tuner module extract the required frequency band and perform analog to digital conversion. The resulting signals are processed by the digital signal processor. The DSP performs the *blind beamforming (ACMA)* [23] to detect and extract individual signals from the input mixture. The individual signals are then *demodulated*. A message is identified using the start flag, end flags and the length of bit stream and CRC. CRC checks are then applied in order to detect any possible transmission errors. Finally the valid messages are extracted.

The Multi-input receiver system shall be optimized for the DSP platform in order to achieve real-time performance in terms of speed and stay within the power budget of the system. The receiver shall work efficiently in the presence of noise and interferences.

1.2 Scope of work

The *Multi-input receiver* is software defined as the key components, the *beamforming* and the *demodulator* are implemented in software.

The scope of this thesis work is,

- Understand the Multi-input receiver system. It includes *blind beamforming* algorithm and *demodulator*.
- Investigate the Multi-input receiver system for performance optimization in terms of speed on an embedded DSP platform.
- Implement the Multi-input receiver system on Texas Instruments 674x DSP platform for an antenna array of four (4) antennas.
- Optimize the Multi-input receiver system to achieve real-time performance for Texas Instruments 674x DSP processor. For the thesis, a reference case has been defined such that the Multi-input receiver shall process each group of overlapping signals (maximum of four (4)) in less than **25 milliseconds** and the power consumption shall be less than **600 milliwatts**.

- Investigate the performance (speed and power consumption) of Multi-input receiver system in the presence of noise and overlapping signals for realistic scenarios.

1.3 Research Objectives

The research objectives are,

- Investigate the Multi-input receiver for performance improvement in terms of number of successful retrievals.
- Investigate the Multi-input receiver (blind beamforming, demodulator) for performance optimization on Texas Instruments 674x DSP.
- Investigate the Multi-input receiver power consumption on Texas Instruments 674x DSP.

1.4 Document Structure

The document is structured as follows: In **Chapter** 2 we discuss general DSP architecture and discuss the target processor for the Multi-input receiver application, the Texas Instruments TMS320C6748 floating point processor. The objective is to understand the DSP architecture for optimization purposes. In **Chapter** 3 we focus on the blind beamforming (ACMA) algorithm. We summarize the work by Alle-Jan van der Veen and Arogyaswami Paulraj in their paper 'An Analytical Constant Modulus Algorithm' [23]. In **Chapter** 4 we focus on optimizing the Multi-input receiver for the TMS320C6748 DSP platform to achieve real-time performance. We discuss the various optimization techniques applied and present our experiments and results. We also look at the power consumption of the system. Finally in **Chapter** 5 we provide our conclusion and future scope for this thesis work.

Digital Signal Processor Concepts

2

Digital Signal Processors (DSP) are special type of processors designed for computational intensive signal processing operations. Applications such as video or speech processing require real-time performance with high computational capabilities. DSPs can provide these performances with low -cost and low power consumption.

Conventional general purpose processors are designed to execute instructions sequentially. Signal processing applications typically have operations which can be executed parallel. DSP architecture is designed to exploit this characteristic of signal processing applications. It is important to understand the DSP architecture in order to optimize an application on the target application. In this chapter we focus on the architecture of digital signal processors.

This chapter is organized as follows. In Section 2.1 we discuss concept of digital signal processing. In Section 2.2 we discuss common DSP algorithms and its mathematical representations. In Section 2.3 we derive the general DSP architecture from the characteristics of signal processing algorithms. We discuss the functional units, memory architecture, addressing modes and hardware pipelining. Finally in Section 2.4 we discuss the target platform for the Multi-input receiver, Texas instruments TMS320C6748 floating point DSP processor.

2.1 Digital Signal Processing

Signal Processing deals with operations on or analysis of signals in discrete or continuous time domain. Examples of signals are sound, images, control system signals, telecommunication signals, electrocardiogram, sonar and radar signals, seismic data, biomedical signals and many others.

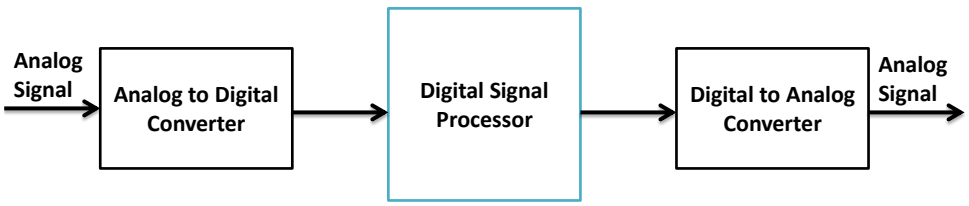

Figure 2.1: A simple digital signal processing system

The traditional method of signal processing is the *Analog Signal Processing*. Analog signal processing is the signal processing performed on analog signals by using analog

components such as resistors, capacitors, inductors, transistors and amplifiers. Analog stands for parameters such as voltage or current is mathematically represented as a set of continuous values. Examples of analog operations are convolution, Fourier transform and Laplace transform. Examples of analog signal processing are bass, treble and volume control filters in HiFi equipments.

Digital Signal Processing (DSP) is an alternate method to process analog signals. In DSP, the analog signals are converted into digital signals. Digital signals are sequence of binary numbers which represents the real world analog signals. They are easy to store in memory and process using a microprocessor. Numerical operations can be easily applied to the digital signal in order to change, enhance or modify the signal. It gives the flexibility to perform complex signal processing tasks much easier than by means of analog circuit. Once the digital signal is processed it can be converted back to analog form.

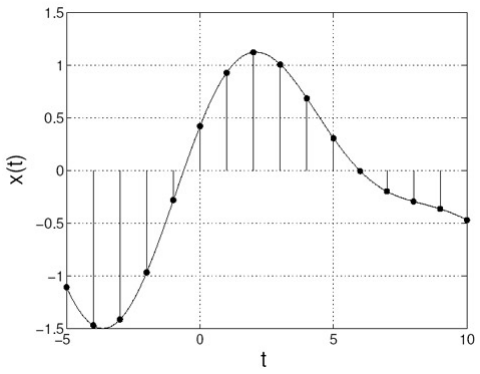

Figure 2.2: ADC: Sampling and approximation

The steps involved in digital signal processing are shown in Figure 2.1. The analog signals are converted into digital domain by an analog-to-digital converter (ADC) circuit. The ADC uses *sampling and approximation* as shown in Figure 2.2, where the analog signals are sampled at specific time instances. The analog samples are measured using approximation circuit and converted into numerical (digital) format. The digital signal is processed by the Digital Signal Processor. Finally the digital signals are converted into analog format by using a digital-to-analog converter (DAC) circuit.

Digital signals can be proceed using general purpose (sequential) processors, but this may result in slower processing rate. Most of the signal processing applications have real-time performance requirement. The *real-time* [15, 17] constraints are based on the application. For example a speech processing application has a signal frequency of 4 KHz. According to Nyquist theory this signal has to be sampled at 8 KHz or higher in-order to produce non-corrupt (non aliased) samples. Let us assume the sampling frequency is 10 KHz. Now the time period between successive samples is 100µs. If there are 100 instructions each instruction must execute in 1µs to meet the real-time requirements. The sampling frequency for music can be for example 48 KHz, Video Phone 6.5 MHz,

TV 27 MHz, and HDTV 144 MHz. As you can see the processing requirements are getting more and more demanding. To address the high performance requirements of DSP applications the digital signal processors have parallel architecture.

2.1.1 Advantages and Disadvantages of DSP

DSP has many advantages over analog signal processing which makes it a popular choice for specific applications.

1. Digital samples are easier to manipulate using hardware and software than traditional analog techniques. It is able to provide better levels of signal processing than possible with analog hardware alone.
2. Digital Signal Processing is more deterministic, robust and flexible.
3. Digital Signal Processing is more reliable as the processing is not affected by the temperature or age of the components.
4. It is easy to update a digital signal processor by updating its software.
5. In Digital Signal Processing we can implement functions which are not possible with analog hardware. FIR filter is an example of a digital signal processing function.

Digital Signal Processing has the following disadvantages;

1. Since the digital samples are approximations of the original analog signal it is not possible to provide the perfect demodulation, filtering or other functions.
2. It can be more expensive than the analog solutions, hence in some cases it may not provide the most cost effective solution.

2.1.2 Classification of DSP

Digital Signal Processing elements can be classified based on its evolution.

- **Discrete logic:** The first generation of digital signal processing tools was based discrete logic gates. They were implemented using bipolar SSI, MSI technology. They were used for non-real time applications.
- **Building block:** The next generation DSPs were based on building blocks such as single chip bipolar multiplier and Flash ADC. They were used for digital communication and military radars.
- **Single Chip DSP:** Single Chip DSPs with its own microprocessor architecture based on NMOS/CMOS technology. They were used in control systems and telecommunications.
- **Function/Application specific chips:** These DSPs are based on vector processing and parallel processing. They find applications in computers and communications.

- **Multiprocessing:** These is advanced multiprocessing DSPs based on Very Long Instruction Word (VLIW) or Multiple Instructions Multiple Data (MIMD) technology. They are used in high end video/image processing.
- **Single-chip multiprocessing:** These are low power multi-processing single-chip DSPs. They find applications in Wireless devices and Portable multimedia devices.
- **SoC multiprocessing:** The advanced DSPs are low power multiprocessing System-on-Chip. They find applications are low power portable devices such as high end mobile phones.

2.1.3 DSP Applications

Digital Signal Processors are widely used in many signal processing applications where high processing is required at low cost and low power consumption. They are designed to process multiple instructions simultaneously. They typically run one program unlike general purpose processors. DSP applications typically have hard real-time constraints.

DSPs find applications in many areas. They are widely used for multimedia application for audio/video processing. They are used in Televisions, Digital Cameras, Mobile phones and other portable devices. They are widely used in Networking and Telecommunication such as cellular telephony. They are used for medical applications such as image processing. They are used in military, industrial control, storage products etc. They are also used for Speech coding, Speech recognition, Speech synthesis, identification high end audio systems, Audio equalization, Noise cancellation, Modems, Ambient Aquatic emulations, Audio/Video editing and mixing, Navigation, Visual systems such as security cameras, Image manipulations, beamforming, Spectral Estimation etc.

2.2 DSP Algorithms

Typical analog signal processing algorithms are convolution, Laplace transform, Fourier transform etc. When the signals are converted into digital domain, digital signal processing algorithms are applied on the digital signals. DSP algorithms are the counterpart of its analog signal processing algorithms. Common DSP algorithms are Discrete Fourier Transforms (DFT), Fast Fourier Transforms (FFT), convolution, correlation, digital filtering (FIR, IIR), searching paths for decision trees in finite state machine (FSM) diagrams etc. The architecture of the Digital Signal Processor and its Instruction Set Architecture (ISA) are derived from the signal processing operations [18]. So it is important to understand the basic structure of signal processing algorithms to design DSP hardware.

Most DSP algorithms such as FFT, convolution, filters can be represented in simple form using *sum-of-product* equation as;

$$Y = \sum_{n=1}^{N} A_n * X_n \tag{2.1}$$

2.2.1 Classification of DSP Algorithms

DSP algorithms can be classified based on their usage and their implementation.

Based on the usage DSP algorithms are classified as

- **Time domain:** Time domain signals are where the information is encoded in the shape of the waveform of the signal. Time domain algorithms are used to process these signals. Operations include DC removal, waveform shaping etc.
- **Frequency domain:** In frequency domain signals the information is encoded in the amplitude, frequency and the phase of the sinusoidal components which constitutes the signal. Frequency domain algorithms process these signals to separate frequency bands from one another.
- **Custom:** Custom algorithms are those which are used for specific requirements other than time and frequency domain operations.

Based on the implementation DSP algorithms are classified as

- **Recursive:** They have a feedback connection from output to the input signal. Example of recursive algorithm is Infinite Impulse Response (IIR) filter.
- **Non recursive:** Non recursive filters have no feedback connection from output to input. Examples of such filters are convolution, Finite Impulse Response (FIR) filter. They have better performance than recursive filters but they are slower in nature.

2.2.2 Examples of DSP operations

Let us consider some common DSP algorithms and its mathematical representations.

Finite Impulse Response (FIR) filter: It is a discrete time filter used for many applications such as speech or image processing. The FIR filter is defined as;

$$y[n] = \sum_{i=0}^{N} h_i * x[n-i] \tag{2.2}$$

$x[n]$ is the input signal $y[n]$ is the output signal, h_i are the filter coefficients, also known as tap weights, and N is the filter order – an Nth-order filter has(N+ 1) coefficients.

Infinite Impulse Response (IIR) filter: IIR is defined as;

$$y[i] = \sum_{k=1}^{M-1} a[k] * y[i-k] + \sum_{k=0}^{N-1} b[k] * x[i-k] \tag{2.3}$$

$x[n]$ is the input signal $y[n]$ is the output signal, a_i,b_i are the filter coefficients.

Discrete Fourier Transform(DFT): It is used for spectral analysis in the frequency domain.

$$y[k] = \sum_{n=0}^{N-1} W_N^{nk} * x[n];\ where\ W = e^{\frac{-2j\pi}{N}} \tag{2.4}$$

$x[n]$ is the input signal in time domain. $y[n]$ is the output signal in frequency domain. for k=0,1,2,...,N-1

Inverse Discrete Fourier Transform(IDFT):

$$x[n] = \sum_{k=0}^{N-1} W_N^{-nk} * y[k];\ where\ W = e^{\frac{-2j\pi}{N}} \tag{2.5}$$

for n=0,1,2,...,N-1

Fast Fourier Transform(FFT): It is an efficient method for finding the DFT.

Discrete Cosine Transform(DCT) and Inverse DCT: Is commonly used in video compression.

$$y[k] = e(k) \sum_{n=0}^{N-1} cos[\frac{(2n+1)k\pi}{2N}] * x[n]; for\ k = 0, 1, 2..N-1 \tag{2.6}$$

$$x[n] = \frac{2}{N} \sum_{k=0}^{N-1} e(k) * cos[\frac{(2n+1)k\pi}{2N}] * y[n]; for\ k = 0, 1, 2..N-1 \tag{2.7}$$

From the above examples it is obvious that the DSP algorithms can be represented in simple *sum-of-product* equations. The basic operations needed to implement sum-of-product equations are multiplication and addition. These operations are repeated a number of times (looped) for each sample. To design DSP hardware one has to provide these basic functionalities along with storage for input samples and coefficients. The main design goal of a Digital Signal Processor is to provide high speed functionality and high throughput with minimal hardware complexity [22].

2.3 DSP Architecture

In this section we discuss the basic digital signal processor architecture and the architecture features in detail.

Special Purpose DSP processors (ASICs): They have specific hardware designed for 1) the execution of specific DSP algorithms or 2) execution of specific DSP applications. Examples of algorithm specific DSP hardware are FFT processor (PDSP16515A) and programmable FIR filters (VPDSP16256). Example of application specific DSP hardware is Mitel's multi-channel telephony voice echo canceler (MT9300).

General Purpose DSP processors: They are high speed general purpose processors with hardware and instruction set optimized for execution of DSP operations. Examples of general purpose DSP processors are Texas Instruments TMS320C64x and Analog Devices ADSP21xxx SHARC processors. In this work focus on general purpose DSP processors.

2.3.1 High level Block diagram

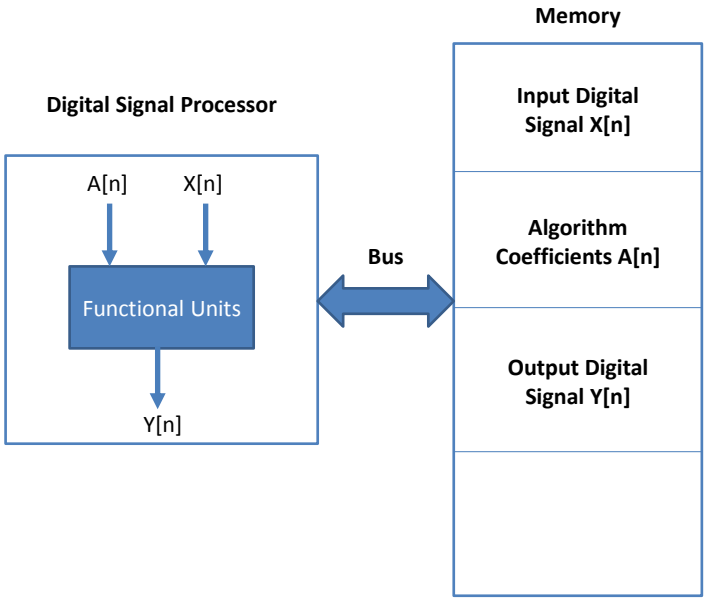

Figure 2.3: High level block diagram of a DSP processor derived from the DSP algorithms

From the previous section we have seen that the DSP algorithms are represented as the sum-of-products equation

$$Y = \sum_{n=1}^{N} A_n * X_n \tag{2.8}$$

Where X[n] is the input signal, A[n] are algorithm coefficients and Y[n] is the output signal in digital format.

The high level block diagram of a DSP processor derived the Equation 2.8 is shown in Figure 2.3. The DSP is interfaced with the memory where the input signals, the coefficients and the output signals are stored. It reads A[n] and X[n] from the memory and calculates the output signal Y[n]. The output is then saved to the memory.

2.3.2 Functional Units

Most DSP algorithms can be represented as *sum-of-products* Equation 2.8. The basic functional unit required for this operation is a hardware *Multiplier* which can multiply two numbers A_i and X_i. Once the multiplication is completed we need an

Adder/Subtracter functional unit to perform the summation. It is typically implemented with an Arithmetic and Logic Unit(ALU) which can additionally perform logical operations like AND,OR and XOR. Now we need *Storage* for holding the variables Y_i, A_i and X_i. We can use internal registers (accumulator) as the storage for temporary variables needed for computation. The easiest way to implement such a DSP hardware is a *Multiplier and Accumulator (MAC)* unit is shown in Figure 2.4.

The multiplication of two n bit numbers will produce a $2n$ bit result. It may cause overflow of the result. To avoid such situation either a wider accumulator could be used or perform a shift and round adder before the ALU as shown in Figure 2.4.

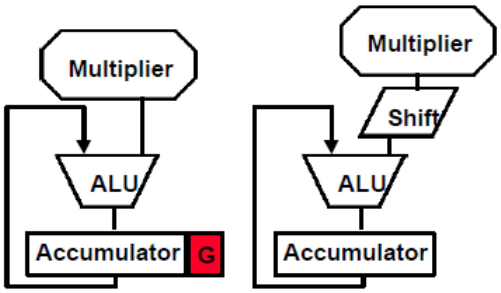

Figure 2.4: Multiplier and Accumulator (MAC) Functional Unit. Wider Accumulator MAC and sift right and round product MAC

The MAC functional unit of the first commercial DSP processor, Texas Instruments TMS32010 is shown in Figure 2.5.

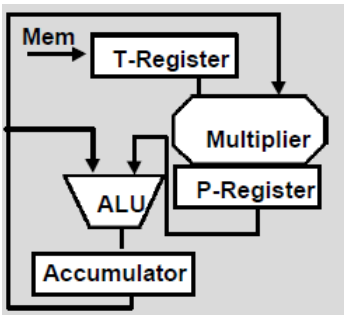

Figure 2.5: Texas Instruments TMS32010 processor

In programing when an operation or a sequence of operations are to be performed a number of times, it is a common practice to use a *loop* mechanism in order to avoid repetitive code. Using a loop helps to reduce the code size and hence to save the valuable program memory. The advanced DSPs [17] provide additional functional unit to perform the *branching/looping* operation. It allows you to set the loop counter and increment or decrement the loop counter through instruction. Based on the value in the loop counter it facilitates *jump* or *conditional jump* to a different location in the program sequence.

Another functional unit commonly seen in commercial DSPs are the *load/store* functional unit. General purpose CPUs supports Complex instruction set computing (CISC) which allows direct addressing mode where we can use a memory address as operands. For example, MOV A,30h instruction copies the content of memory location 30h to the accumulator. Most of the advanced DSPs uses *load/store* Reduced instruction set computing (RISC) Architecture [17,19]. In this operations are performed on the data in the following way;

1. Load data from the memory to the internal registers.
2. Perform operations on the data stored on the internal register using functional units.
3. Store the result from internal register to the memory.

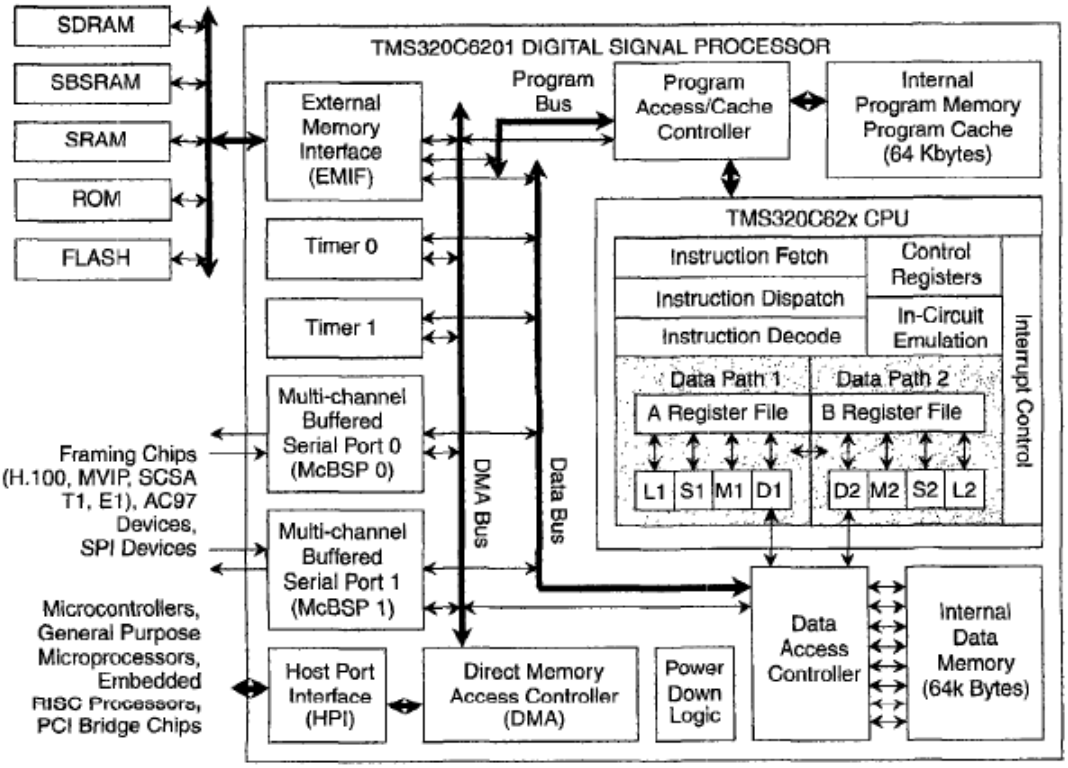

Figure 2.6: Functional Units in advanced load/store RISC based DSPs. TI TMS320C6x Processing Unit [17]

The load and store operations are performed using indirect addressing mode(using pointers) The advantage of load/store based RISC architecture is that it makes the hardware simpler and programming easier and efficient. Texas Instruments DSP processors and Analog Devices Blackfin processors are examples of RISC DSP processors.

The advanced DSPs have separate functional units to perform various operations instead of conventional MAC unit. For example the functionals unit for an advanced Texas Instruments TMS320C6x processor is shown in Figure 2.6. It has 8 functional units. In this diagram M1, M2 are the *Multiplication* units, L1, L2 are the *ALUs*, S1, S2 are the *Branching* units and D1, D2 are the *Load/Store* unit. All these functional units are capable of performing 32 bit operations. A and B are two *Register files*, each containing 16, 32-bit registers. Such architecture is capable of executing up to 8 instructions in a single clock cycle.

DSP functional units can be implementation as fixed point or floating point. Fixed point operations are faster than floating point operations as it requires less hardware resources (logic gates). Functional units may support additional features like modulo arithmetic, saturation, rounding, single instruction multiple data operations. More details about the TI TMS320C6x DSP platform are given in Section 2.4.

2.3.3 DSP Architecture Features

In this section we discuss the architecture features of digital signal processors. We discuss the memory architecture in DSPs, the specialized addressing modes, specialized instructions and hardware pipelining used in DSPs.

2.3.3.1 Memory Architecture

We have seen from the DSP architecture that they are designed to perform multiple operations simultaneously. One of the major bottlenecks in DSP processor is transferring large amount of data (program instruction and operands) to and from the memory to cope up with the computation speed [19]. For example if we want to multiply two numbers we need to fetch the program instruction as well the two operands. If we are executing 8 such instructions in parallel, it needs to fetch 8 instructions and 16 operands at each cycle.

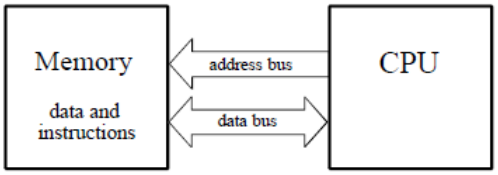

Figure 2.7: Von Neumann architecture; single memory for instructions and data [19]

The traditional Von Neumann architecture [19] is shown in Figure 2.7 has common instruction and data memory. It is a satisfactory solution when instructions are executed in serial fashion. For parallel execution of instructions as in DSPs, the Harvard Architecture [19] shown in Figure 2.8 provides better bandwidth to the memory. It uses separate instruction(program) and data memory and with separate bus connections to

each. This dual bus architecture allows the processor to fetch program and data at the same time. Most modern DSPs use the dual bus architecture.

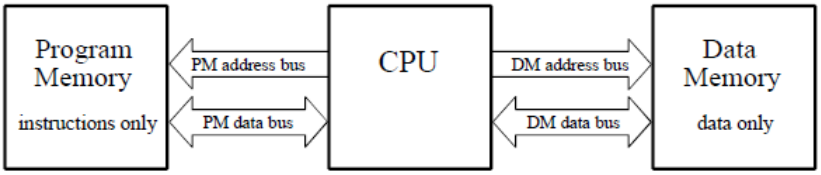

Figure 2.8: Harvard Architecture; Separate memory for instructions and data [19]

2.3.3.2 Addressing Modes

CPUs have several addressing modes supported by their instruction set architecture (ISA). Addressing modes define how the processing unit can reach an *operand* defined in an *instruction* in its machine language. The addressing mode specifies how to calculate the effective memory address of an operand.

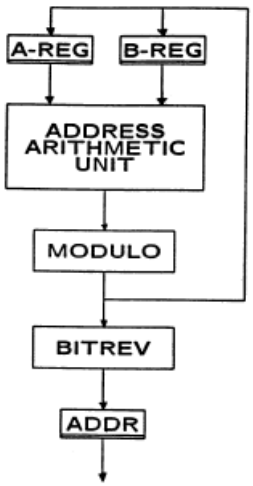

Figure 2.9: Address Calculation Unit for DSPs

Typically general purpose processors support many addressing modes, where a DSP processors support a limited set of addressing modes. DSPs typically implement dedicated address generation Units. In this section we discuss briefly the various addressing modes supported by the DSPs.

The most common addressing modes supported by DSPs are *immediate* and *displacement* addressing modes. MOV A,#20h is an example of immediate addressing. This instruction stores the immediate value mentioned to the register A. Add R4, 100(R1)

is an example of displacement addressing mode where the content of memory address R1+100 is added with content of register R4.

Another most common addressing mode supported by DSPs are *register indirect*. In load/store architectures [17] register indirect addressing is the only way to access memory. In this data is loaded from the memory using pointers and stored in its internal register. Operations are performed on the data saved in its internal register. The result is stored back to the memory from internal register using pointers.

Auto-increment/Auto-decrement register indirect is another addressing mode supported by DSPs. For example in LW R1,0(R2)+ loads the content of memory at address stored in register R2 to register R1 and increments the pointer R2 to address the next memory location. This addressing mode is very effective to access data inside a loop.

The *bit-reversed* addressing mode to access *circular buffers* [19] are another common addressing mode in DSPs. Some algorithms such as Infinite Impulse Response (IIR) filters perform better in multiple stages. Multiple stages require circular buffers to store intermediate data. When DSPs are used for Fast Fourier Transforms (FFT) the *bit-reversed* addressing mode into the circular buffers provides faster execution of the algorithm. In bit reversed addressing mode subsequent address are generated by reversing the address bits from LSB to MSB. Another special addressing mode supported by DSPs are the *modulo(circular) addressing* mode. A typical address calculation unit for DSPs is shown in Figure 2.9. It supports modulo and bit-reversed addressing. It is often duplicated to calculate multiple addresses per cycle.

2.3.3.3 Special Instructions

In this section we discuss the typical DSP instructions. In DSPs the most common instructions are Multiplication and Addition. All DSPs support multiplication as a single instruction instead of being implemented using other instructions. Addition/Subtraction is supported through ALUs which can perform logical operations such as AND, OR, XOR etc. Most DSPs support MAC instructions which are the basic element in DSP algorithms.

DSPs have branch/loop instructions typically supported by the hardware to reduce the branch loop latency. Conditional branching instruction is commonly supported by DSPs. They also support saturated shift operation arithmetic.

DSPs support specialized complex instructions [17] to support signal processing algorithms. For example a 32 bit adder could perform two 16-bit addition or four 8-bit additions to make the computation faster for applications which has data width of 16-bit or less.

2.3.3.4 Hardware Pipelining

In-order to improve the performance DSPs provide parallel hardware. Typically DSPs are capable of fetching, decoding and executing multiple instructions in parallel. For

example the Texas Instruments TMS320C6x architecture has 8 functional units, which means it can fetch 8 instructions, decode them and execute them simultaneously. One of the most important features in a processor to speed up the execution is *pipelined architecture*. In this section we discuss the concepts of Pipelining and explain how it is used in DSPs.

2.3.3.5 Pipelining

The basic stages involved in executing an instruction are 1) Instruction fetch (F) 2) Instruction Decode (D) 3)Instruction Execute (E). Let us assume that each step takes one cycle to execute.

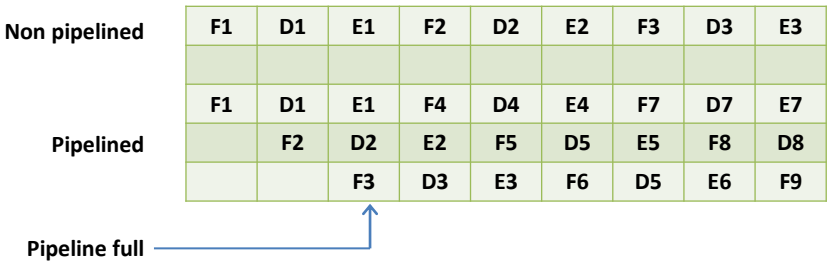

Figure 2.10: Concept of Pipelining

Conventional processors perform all these steps in a serial (non-pipelined) fashion as shown in Figure 2.10. An instruction is fetched, decoded and executed and the steps are repeated for the next instruction. It is called the non-pipelined architecture. As you can see each instruction takes 3 one cycle to execute. If there are 10 instructions it will take 30 clock cycles to complete the execution.

Modern processors uses parallel hardware (separate functional units, memories and buses) which enables it to overlap these stages in time as shown in Figure 2.10. For example in the 1st clock cycle the first instruction is fetched. In the 2nd clock cycles while the the first instruction is decoded, the second instruction can be fetched. From third cycle onwards one instruction can be fetched, decoded and executed in every clock cycle. This is called the pipelined architecture where several instructions can be in different stages of their execution cycles. The pipeline is *full* when all the stages are operating simultaneously. The benefit of the pipelined architecture is that when the pipeline is full each instruction takes only one cycle to execute. To execute 10 instructions, it takes 12 cycles instead of 30 cycles.

The advantage of pipelining is faster execution. The disadvantage is that whenever there is a branching or interrupt the pipeline has to be flushed; discarding the already fetched instruction and start over again with new fetch.

2.3.3.6 Pipelining in DSPs

Pipelining in DSPs is complex than its definition in the previous section. In this section we explain how the pipelining is implemented in a Texas Instruments TMS320C6x VelociTI [17, 19] fixed point processor architecture.

The basic phases in pipelining fetch, decode and execute are divided into stages and each stage is pipelined.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
S1	PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	PG	PS
S2		PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	PG
S3			PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6
S4				PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5
S5					PG	PS	PW	PR	DP	DC	E1	E2	E3	E4
S6						PG	PS	PW	PR	DP	DC	E1	E2	E3
S7							PG	PS	PW	PR	DP	DC	E1	E2
S8								PG	PS	PW	PR	DP	DC	E1
S9									PG	PS	PW	PR	DP	DC
S10										PG	PS	PW	PR	DP
S11											PG	PS	PW	PR
S12												PG	PS	PW

Pipeline Full

Figure 2.11: Advanced Pipelining used in TI TMS320C6x Architecture [19]

The program fetch phase consists of 4 stages,

- PG: Program address generates. It computes the address of the next fetch packet.
- PS: Program address sends to memory.
- PW: Wait for the memory to be ready to send the fetch packet.
- PR: Program fetch packet receives.

The decode phase has 2 stages,

- DP: Instruction dispatch. TI uses a concept of fetch packets where each packet consists of up to 8 instructions. Fetch packets are separated into execute packets. The instruction of execute packets are routed to the decode unit of the appropriate functional unit.

- DC: Instruction decode. In this stage each instruction is decoded.

The execution phase is divided into 6 stages (10 stages for TMS3206X floating point architecture). Each instruction requires a fixed number of stages to complete the execution. The 6 execution stages link directly to the latency involved in executing each instruction. For example an addition can be performed in 1 cycle without any latency. A multiplication instruction has a latency of 1 cycle and it takes 2 cycles to complete the execution. Similarly a load instruction has a latency of 4 cycles and branch instruction has a latency of 5 cycles.

In the TMS320C6x VelociTI Architecture [17] the pipeline has 12 stages consist of 4 fetch stages, 2 decode stages and 6 execution stages as shown in Figure 2.11.

2.4 Target Platform: TI 674x DSP

So far in this chapter we have studied the generic architecture of DSP processor and how they are derived. We have also looked into the processing units of Texas Instruments TMS320C6x processor and its pipeline organization. In this section we provide some information specific to the target platform for our Multi-input receiver application.

The target platform chosen for our application is Texas Instruments TMS320C6748 VLIW DSP core. It is a floating point DSP processor, compatible with 64x+ fixed point DSP processor. It means that the 674x core can execute both floating point and fixed point instruction. From the optimization perspective, sometimes fixed point arithmetic can provide better performance. The processor can run at 375/456 MHz clock frequency.

The block diagram of the target platform is shown in Figure 2.12 [25]. The basic functional units of the system are shown in Figure 2.6. The 674x DSP Central Processing Unit(CPU) consist of 8 functional (execution) units, two register files and two data paths. The two general purpose register files A and B contain thirty two 32-bit registers each. It can be used for storing data as well as data address pointers (674x DSP has load/store architecture). The data types supported are packed in 8 bit, 16 bit, 32 bit, 40 bit and 64 bit data.

The eight functional units (.M1, .L1, .D1, .S1, .M2, .L2, .D2, .S2) are capable of executing one instruction every clock cycle. .M functional unit perform the multiplication operations. .L performs all the arithmetic and logical operations. .S performs the branching operations. .S can also be used for arithmetic and logic operations. .D is used for loading and storing data between register files and memory.

674x DSP can perform both fixed point and floating point arithmetic.

The capabilities of .M units for performing floating point operations on single precision (SP) and double precision (DP) arithmetic are,

- Two SP x SP to produce SP in every clock cycle.

- Two SP x SP to produce DP in every two clock cycles.
- Two SP x DP to produce DP in every three clock cycles.
- Two DP x DP to produce DP in every four clock cycles.

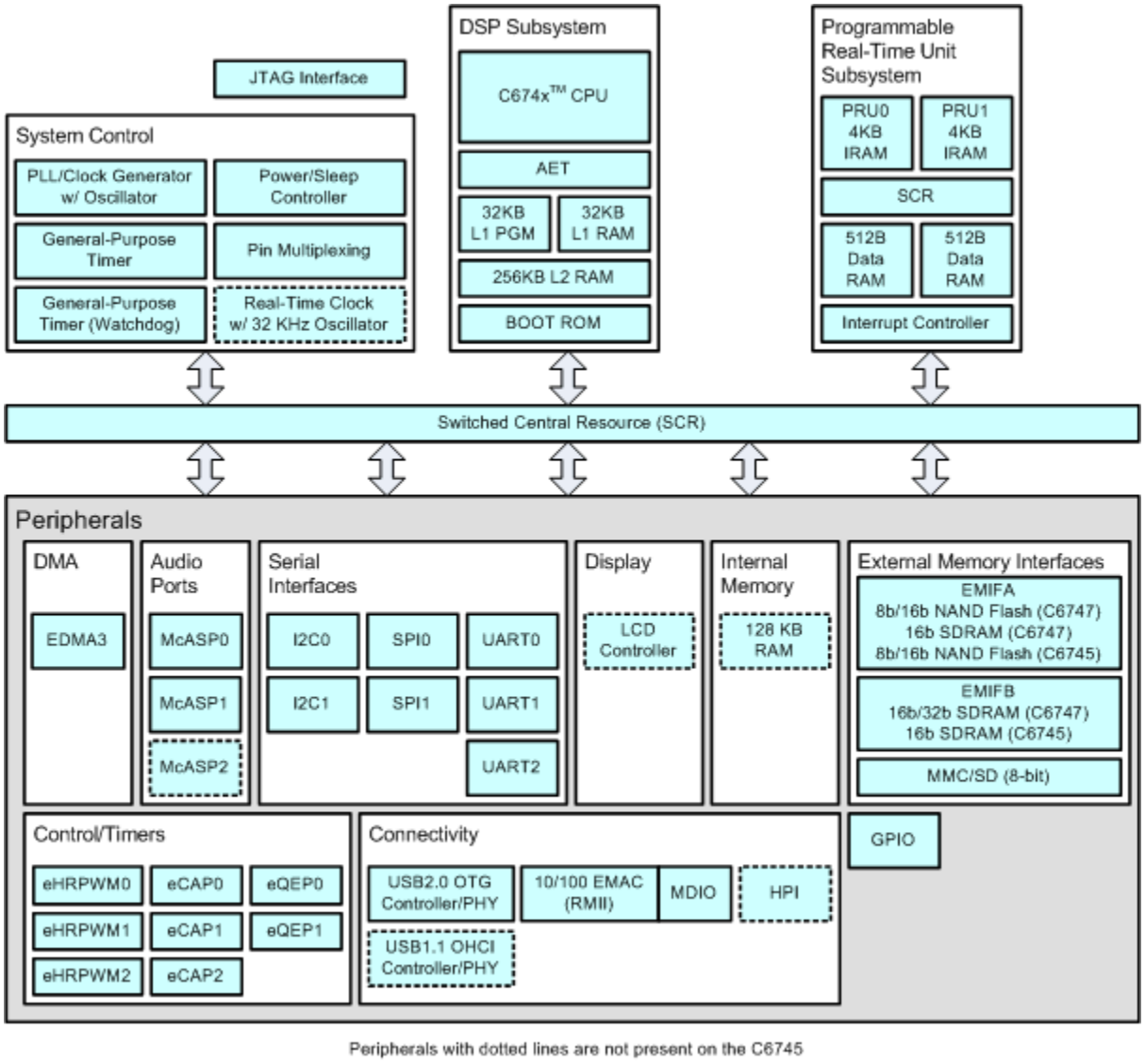

Figure 2.12: Functional Block Diagram of the target platform - TI 674x DSP [25]

The .M unit can perform the following fixed point arithmetic in one clock cycle,

- Two 32 x 32
- Four 16 x 16
- Eight 8 x 8

The .L unit can perform parallel operation on each execution units. For example it can perform one 32 x 32 or two 16 x 16 or four 8 x 8 arithmetic and logic operation on each execution unit per cycle.

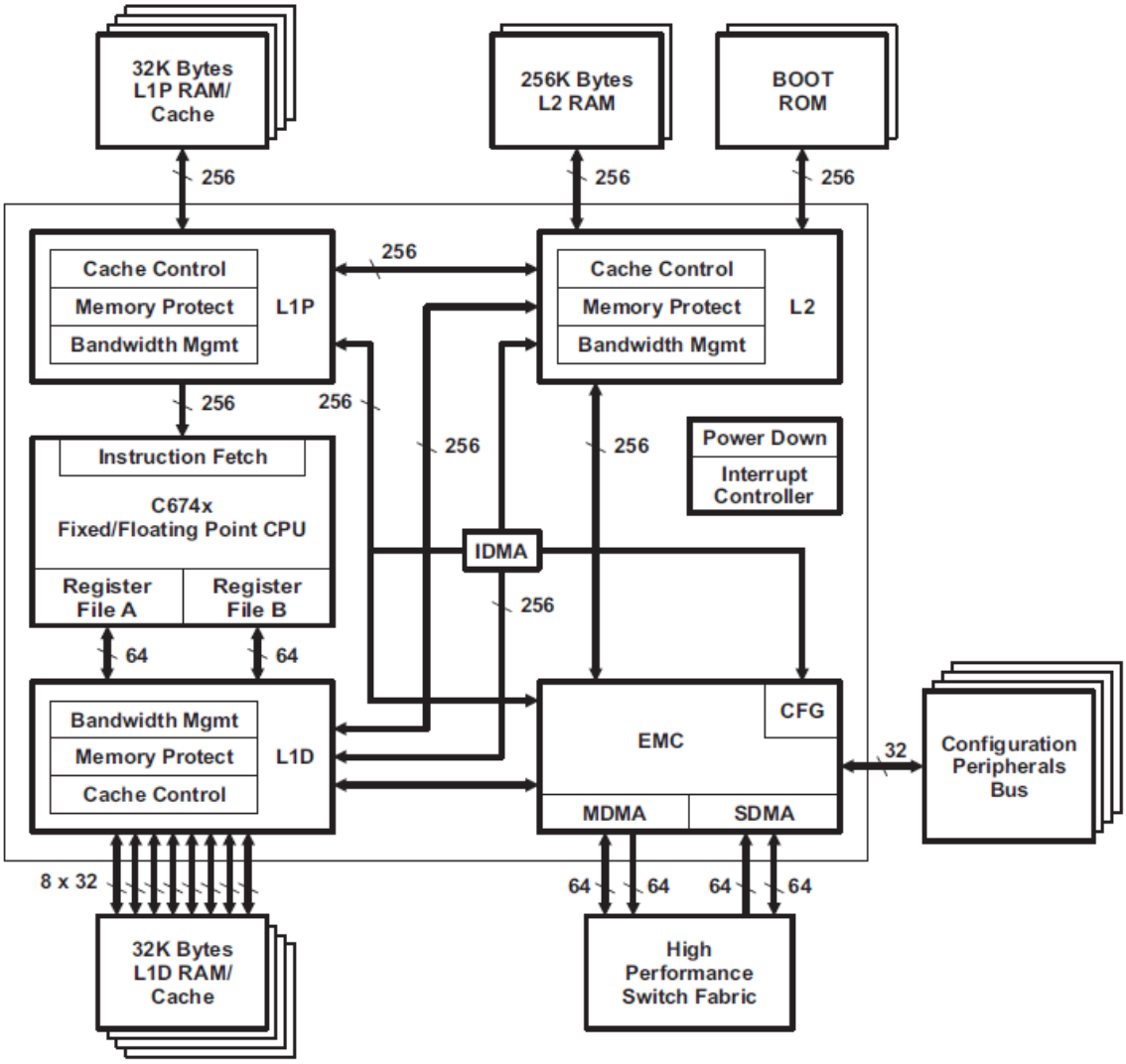

Figure 2.13: Internal block diagram of TI 674x DSP [24]

The .S unit supports *SPLOOP*, a small instruction buffer in the CPU which helps the software pipelining of loops. In software pipelining method multiple iterations of the loop are executed in parallel.

The DSP subsystem and the internal memory bandwidth are shown in Figure 2.13. The features of the C674x DSP CPU subsystem are,

- 32KB L1 Program (L1P)/Cache (32KB).

- 32KB L1 Data (L1D)/Cache (32KB)
- 256KB Unified Mapped RAM/Cache (L2)
- Boot ROM (cannot be used for application code)
- Little endian

2.5 Summary

In this chapter we have discussed the Digital Signal Processor concepts. We have focused on the design of DSP hardware. We have seen that DSP hardware is derived from basic DSP signal processing algorithms. The core processing unit of a DSP is a *multiplication and accumulation* (MAC) unit. The rest of the DSP architecture is designed to keep the MAC unit busy. In order to achieve the high computation requirements of todays applications, DSPs adapt multiple functional units, multiple memories and buses to perform operations in parallel. It also uses specific addressing modes to suit faster signal processing algorithm implementations. It efficiently uses hardware pipeline for its advantage. Additionally advanced DSPs make use of extended parallelism (Single instruction, multiple data (SIMD) processing, and Very-long-instruction-word (VLIW) processing) to achieve increased computational performance. Todays DSP provide low-cost solution, high performance, low power consumption and low latency, which makes them an ideal co-processor for portable devices. Understanding of DSP processor architecture is very important when optimizing the code for performance on a given target platform.

An Analytical Constant Modulus Algorithm

3

Many communication signals have the constant modulus (CM) property. Frequency modulation (FM), Phase modulation (PM), Frequency-shift keying (FSK), Phase-shift keying (PSK) are example of constant modulus signals. If the signals are corrupted by noise/interference, the CM property is lost. Constant modulus algorithms (CMA) can restore this property by finding a filter W and without having knowledge of the sources. In this chapter we focus on a new approach an **Analytical Constant Modulus Algorithm (ACMA)**.

This chapter we briefly summarize the work of Alle-Jan van der Veen and Arogyaswami Paulraj [23]. In Section 2.1 we describe the beamforming problem for constant modulus co-channel overlapping signals. We also briefly describe the generic solutions using Constant Modulus Algorithms (CMA) to separate out individual signals. In Section 2.2 we describe the proposed ACMA algorithm. Finally in Section 2.3 we discuss the advantages of ACMA algorithm which makes it suitable for our application.

3.1 Beamforming

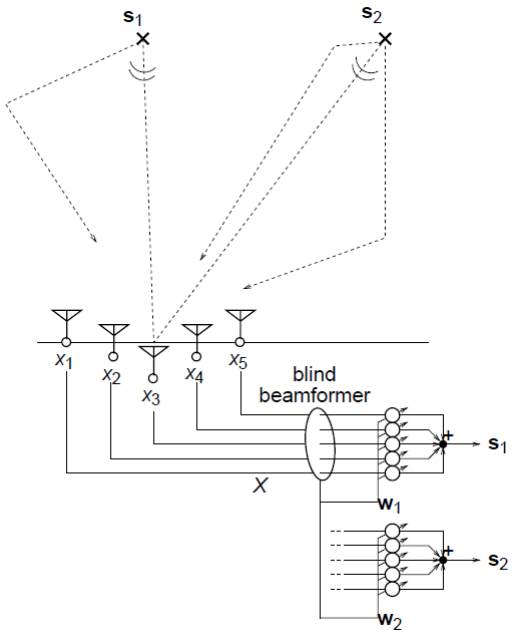

Figure 3.1: Blind beamforming scenario [23]

Consider a number of sources at distinct locations transmitting signals at the same frequency and at the same time as shown in Figure 3.1. It causes co-channel interference of the signals. Beamforming is a method used in spatial signals processing to receive individual signals negating the interferences based on its direction of arrival. It requires an antenna array to receive the signals. Beamforming is done by means of computing a weight vector $\mathbf{w}_i$. If the weight vectors are solved only from the measured antenna array data (without any information about the signals, channel or direction of arrival) it is called **blind beam forming** [23].

This scenario can be mathematically represented as,

$$\mathbf{X = A\ S,}$$

Where the matrix X contains n samples from the m antennas. X has a dimension of m x n. A is the array response matrix of size m x d, where d is the number of signal sources. S is a d x n source signal matrix. Each row $\mathbf{s}_i$ corresponds to signal from respective source. This model describes the stationary propagation environment as the multiple source signal paths have negligible delay. The beamforming problem can now be visualized as structured matrix *factorization problem*; given X, find the factors A and S satisfying certain structural properties. Once A is known we can calculate $W = A\dagger$, the pseudo-inverse of A. The weight vectors $\mathbf{w}_i$ is given by the rows of the matrix W [23].

Constant Modulus Algorithms (CMA) [20, 23] are typically used to solve the blind beamforming problem for multi-input data.

The CMA algorithms have several disadvantages in the retrieval of all the individual signals present in the channel. The CMA algorithm can filter out one of the interfering signals rather than all the desired signals. To capture more than one signal, a multi stage (iterative) CMA is required. In a multi-stage CMA implementation, after detecting the first signal, it is removed from the source signal to detect the second signal. This process is repeated to retrieve more source signals. The number of signals retrieved by CMA algorithm is limited by the number of antennas. Moreover they are noisy and slow.

3.2 Analytical Constant Modulus Algorithm (ACMA)

A new approach was proposed for constant modulus factorization problem [23]. It is treated as a generalized Eigenvalue problem and is solved analytically.

This method uses a deterministic algorithm and uses only a modest amount of sample from the antenna array. For $d <= m$ sources, and without noise, $n > d^2$ samples are sufficient to compute A and S exactly. For $n > d^2$, it is possible to detect the number of CM signals present in X. With X distorted by additive noise, a generalization of the algorithm is robust in finding S, even when n is quite small.

"The algorithm is derived by assigning the weight vector $\mathbf{w}$ such that $\mathbf{w}X$ is a constant modulus signal. This gives n quadratic equations in the entries of $\mathbf{w}$ are linearized

by writing in terms of the *Kronecker product* $\mathbf{w} \otimes \mathbf{w}'$. The Kronecker product is a vector with d^2 entries (**w'** is the complex conjugate of $\mathbf{w}$). If d^2, then the dimensionality of the solution space of this linear system of equations indicates how many constant modulus signals are present in X. Most solution vectors of the linear system do not have the Kronecker structure $\mathbf{w} \otimes \mathbf{w}'$. The core of the constant modulus problem is to find those which possess a Kronecker structure $\mathbf{w} \otimes \mathbf{w}'$. This problem can be transformed into a generalization of an eigenvalue problem by the simultaneous diagonalization of a number of matrices. In the absence of noise, this problem has an essentially unique solution which can be found using standard linear algebra tools. In the presence of noise an approximate simultaneous diagonalization can be be found" [23]. The paper [23] propose an algorithm which shows quadratic convergence.

The number of signals detected by ACMA is limited by the number of antennas.

3.3 Advantages of ACMA

In this section we discuss the advantages of ACMA algorithm which makes it well suitable for our multi-input application.

Typical CMA algorithms can extract only one signal. To extract more than one signal, a multi-stage (iterative) CMA implementation is required. This process is iterative and requires more processing. Moreover this multi-stage approach has practical limitations in terms of number of signals it can retrieve. ACMA algorithm can extract all the signals in single iteration.

ACMA algorithm requires only a modest amount of antenna array input samples. In idle case (in the absence of noise) $n > d^2$ (d is the number of source signals) samples are required to calculate the input signals. Even when the input signal is distorted by addictive noise, the algorithm is robust for small values of n.

ACMA is a block algorithm applied to a collection of samples. It does not process sample by sample which saves computations. It is a blind beamforming algorithm, which does not require antenna calibration or information about the signals or channels. It is robust in the presence of noise. In short the ACMA algorithm can provide a simple solution to the beamforming problem.

3.4 Summary

In this chapter we have studied the beamforming problem to separate out same frequency overlapping signals using an antenna array. The CMA algorithms, which are used to solve the blind beamforming problem have many disadvantages. The ACMA algorithm describes an efficient way to solve the blind beamforming problem using an analytical approach. This new approach has may advantages which makes it a good candidate for our multi-input problem.

Mapping, Optimization And Results

4

The Multi-input receiver application extracts partially and fully overlapping co-channel constant modulus signal using the *blind beamforming (ACMA)* algorithm. The application was first successfully simulated in MATLAB to prove the concept of blind beamforming. In this chapter we discuss the mapping of Multi-input input receiver application using software on an embedded DSP platform. We use an antenna array of four (4) antennas as the front end, which allows detecting up to four signals. We focus on various optimization techniques to achieve real-time performance on the selected DSP platform.

In Section 5.1 we briefly describe our intended Multi-input receiver application and more importantly the performance requirements on the target DSP platform. In Section 5.2 we justify our selection of the target platform, the Texas Instruments TMS320C6748 floating point DSP processor. In Section 5.3 we describe the implementation process of the Multi-input receiver application on the target platform. In Section 5.4 we explain the various optimization techniques applied to archive real-time performance. We present our experiments and provide the results and analysis. We also study the power consumption of the system.

4.1 Application

The input data packet consists of a binary bit stream (message), Cyclic Redundancy Check (CRC-16) and start & end flags (8 bit). The bit streams (messages) are of fixed length. CRC are added to this bit stream to detect accidental errors occurring during transmission. A start flag and end flag are added to this in order to detect a data packet. The complete packet is then modulated using a constant modulus modulation scheme. The signals containing distinct messages are transmitted from different sources at the same frequency and at the same time. This results in partial or full overlap interference of the signals.

The Multi-input receiver is a software defined radio which can successfully extract individual signals negating the signal interferences and noise. It has three basic building blocks; *antenna array, tuner module* and *digital signal processor*. The front-end is an antenna array and the analog tuner module. The tuner module extract the required frequency band and perform analog to digital conversion. The resulting signals are processed by the digital signal processor. The DSP performs the *blind beamforming (ACMA)* [23] to detect and extract individual signals from the input mixture. The individual signals are then *demodulated*. A packet is identified using the start flag, end flags and the length of bit stream and CRC. CRC checks are then applied in order to detect any possible transmission errors. Finally the valid messages are extracted.

The length or duration of a data packet is referred as a *slot*. The number of signals extracted by the ACMA algorithm is limited by the number of antennas in the antenna array [23]. Our current implementation uses an antenna array of four (4) antennas which can be easily extended and configured using software. As a result the Multi-input receiver described in this thesis can detect up to a maximum of four (4) packets per slot.

4.1.1 Requirements

The Multi-input receiver implementation on the embedded DSP platform shall adhere to the following requirements.

- **Execution Speed:** The Multi-input receiver shall consume less than 25 ms per slot. Which means it shall perform beam forming and demodulation up to four signals in that time period.
- **Power:** The power budget of the target platform is limited to 600 mW.
- **Input:** The Multi-input receiver needs to be tested for realistic inputs. The realistic input consists of signal interferences and noise. The major factor contributing to the noise is the 'thermal noise' (Johnson-Nyquist noise). It is the electronic noise generated by the thermal agitation of the electrons inside the electronics components of the receiver such as the antenna arrays, front end amplifiers and more. Noise figure which is the degradation of the signal-to-noise ratio (SNR) caused by components in a radio frequency (RF) signal chain, can also affect the input signal. Signal interference can vary from a few to hundred of signals.

4.2 Embedded Platform

The receiver system is a software defined radio. The target processor chosen was Texas Instrument TMS320C6748 floating point DSP processor. The development platform chosen was OMAP L-138 [6]. OMAP L-138 consists of an ARM9 processor and Texas Instruments TMS320C6748 (674x) DSP processor implemented on a single chip. The functional block diagram of the OMAP L-138 is shown in Figure 4.1. We have used only the DSP core for our application. The major reasons for choosing the TI 674x DSP platform were:

- The receiver system requires high computation performance and low power consumption. The performance constraint dictate each packet to be processed and demodulated in less than 25 ms. The DSP processor can execute up to 8 instructions in parallel at 375/456 MHz, which is expected to provide sufficient computational power for the application.
- The power constraint dictates that the processor shall not consume more than 600 mW. The Power consumption of TI 674x ranges from 11 mW deep sleep mode power to 470 mW total power in full speed. This value gives the power consumption of the processing cores and internal memory. Most practical embedded applications requires external memory access and peripherals. Since external memory access

consumes significant power [1] we would like to keep it as low as possible. In order to save power we have clocked the processor at 300 MHz.

- In our system the received signals are complex single precision floating values. The TI 674x has native support for single precision and double precision floating point instructions. Moreover this processor is compatible with fixed point instructions of TI 64x+ processors. Some part of the algorithm could be implemented using fixed point arithmetic to gain faster implementation when needed.

- Large internal memory. The system has 128 KB on chip SRAM memory. It has two level cache hierarchy; 32KB of level 1 (L1) program cache and 32KB of L1 data cache and 256 KB of shared L2 program and data cache. L2 can also be used as SRAM memory.

- Dedicated hardware support for custom optimizations. For example, software pipeline loop (SPLOOP) buffers for optimizing loop performance.

- A rich set of peripherals: I^2C, SPI with DMA support. Internal and External DMA for fast memory or peripheral data transfer.

- Good development environment (Code Composer Studio (CCS) Integrated Design Environment [3, 21]).

- Availability of highly optimized DSP libraries.

4.3 Implementation

The starting point of this thesis work was a reference implementation of Multi-input receiver on MATLAB and WINDOWS using C. The scope of the work involved:

- Improve the Windows reference implementation in terms of number of detected messages and optimize it for speed.

- Map the Windows reference implementation to TI 674x DSP processor.

- Optimize the DSP implementation to achieve the desired real-time performance in terms of power and speed, without compromising on signals recovery and demodulation performance.

The first step was to improve the Windows reference implementation for performance by using generic C code optimization techniques [8]. The beamforming and demodulation functions were improved for speed by rearranging statements and expression and improving locality of reference .

Next step was to map the Multi-input receiver application to the TI 674x DSP platform. Embedded systems have limited resources in terms of memory and peripherals. It

is a common practice to share resources among various tasks in embedded systems. Dynamic memory allocation is a technique used in programming to allocate storage during run-time of the program. In order to increase the reliability of an embedded system it is desirable to use static memory allocation techniques where memory is allocated during compile time. The Windows reference implementation was based on dynamic memory allocation. It was modified to use static memory allocation for the DSP platform.

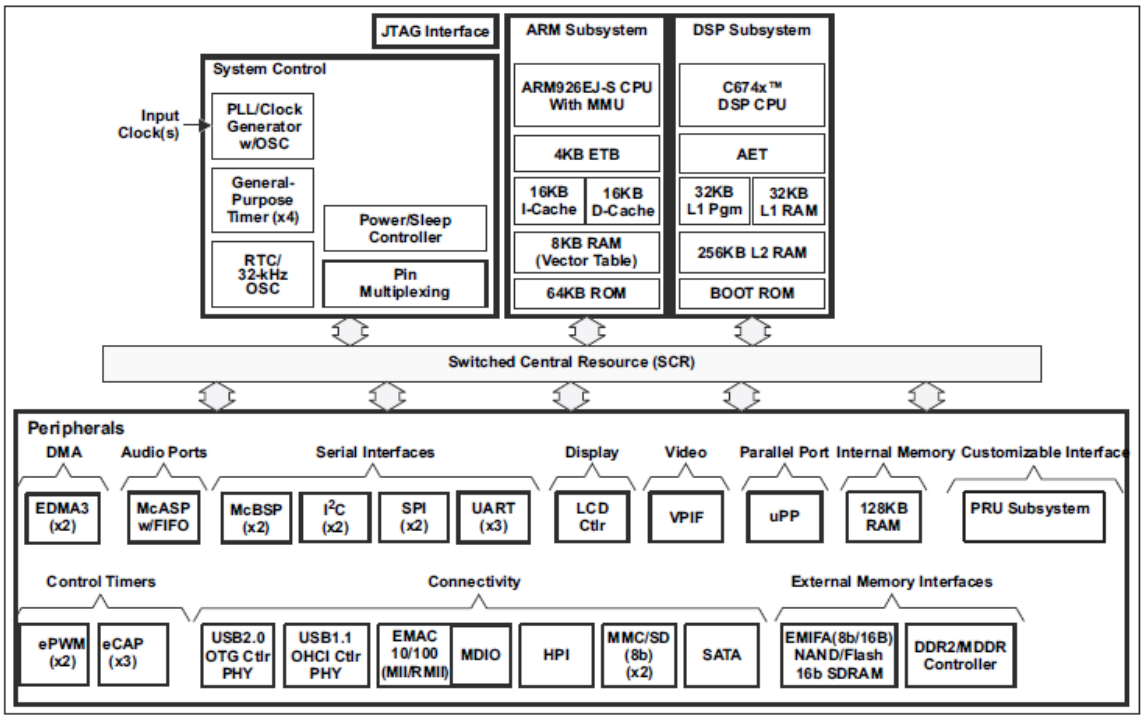

Figure 4.1: Functional block diagram of OMAP-L138 platform [6]

The Windows reference implementation uses double precision floating point arithmetic. Single-precision floating-point values are 32-bit values which can be stored in a single 32 bit register. Double-precision floating-point values are 64-bit values which needs a register pair for storage. Since single precision arithmetic was sufficient for our application the Multi-input receiver algorithm was converted into single precision. It saved required memory for data storage and provide faster execution as single-precision instructions are faster than double-precision instructions.

LAPACK [5, 12] is a software library for numerical linear algebra. It provides routines to solve system of linear equations, linear least squares, Eigen value problems, singular value decomposition and matrix factorizations (LU, QR, Cholesky and Schur decomposition) and more. The beamforming algorithm uses the complex *singular value decomposition (svd)* function from the LAPACK library. The receiver application also uses various matrix factorizations routines from the LAPACK library. LAPACK was initially written in FORTAN and later ported to C programming language (CLAPACK).

The CLAPACK routines depends on Basic Linear Algebra Subprograms (BLAS) and Fortran to C conversion (F2C) software libraries.

LAPACK is a freely-available software package for various platforms. There is no free available implementation of LAPACK for TI 674x DSP. To prove the functionality of the Multi-input receiver on the DSP platform, we needed to port the CLAPACK, BLAS and F2C software libraries into DSP platform. Once we had integrated the ported CLAPACK, BLAS, F2C libraries to the receiver application, we could prove that the receiver system on the embedded DSP platform could successfully separate and demodulate signals. The test inputs were obtained from a scenario simulator tool which generated realistic signals. The receiver output on the embedded platform was tested thoroughly for receiver output correctness.

We could observe that the embedded DSP implementation has a better performance compared to the MATLAB reference implementation in terms of number of successful separation and demodulation of signals. For example the Table 4.1 gives a comparison of performance of two test inputs. The table provides the number of successful retrieval of packets for MATLAB, Windows and DSP (unoptimized) implementations. The packets per slot gives the number of input interfering signals.

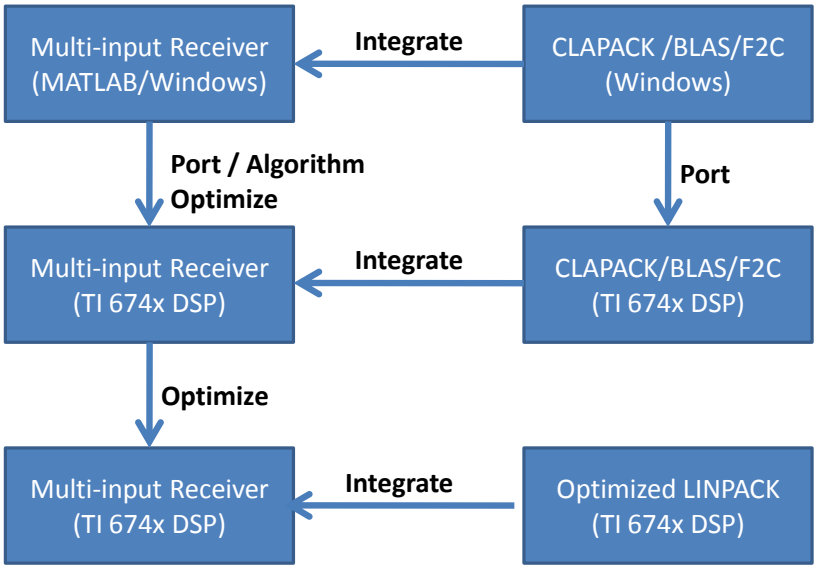

Figure 4.2: Multi-Input receiver Implementation process on 674x DSP (top-down)

We noticed that the receiver application is not fast enough to perform the operations in real time as it consumed 2.67 seconds per slot. The bottleneck for this application was the ported unoptimized CLAPACK and BLAS routines. It was observed that the *singular value decomposition (svd)* function was computation expensive and was required frequently by the blind beamforming algorithm. It was consuming 93 ms per call for an input matrix of size 4x100 on the DSP platform. Since optimizing CLAPACK, BLAS

Input	MatLab	Windows (C)	TI 674x DSP
80 slots (2 packets per slot)	80	91	104
501 slots (1 packet per slot)	305	338	364

Table 4.1: The number of successful retrieval of messages for various implementations (up optimized). The packets per slot gives the number of input interfering signals.

	CLAPACK ported to TI DSP	LINPACK library
Average time per packet slot	2.67 seconds	667 milliseconds

Table 4.2: Average time consumed per slot for Multi-input receiver integrated with ported CLAPACK and optimized LINPACK libraries.

and F2C libraries were out of the scope of this work they were replaced by third party licensed LINPACK (Linear Algebra Package) [4] library assembly optimized for TI 674 x DSP platforms. The observed *svd* timing for the input matrix of size 4x100 was less than 2.5 ms. Table 4.2 provides the comparison of the performance of the Multi-input receiver system integrated for LAPACK ported to DSP and the third party optimized LINPACK library.

We could observe that the performance of the Multi-input receiver on the DSP platform has improved significantly with the replacement of ported CLAPACK libraries by the optimized LINPACK library. The packets in each slot could be processed in 667 ms instead of 2.67 seconds. At this point we need improvement in speed by a factor of 27 to achieve the real time performance (<25 ms). Various optimization techniques are performed on the algorithm as described in the next section to achieve the desired results. The steps involved in mapping the Windows reference implementation to the 674x DSP platform are depicted in Figure 4.2.

4.4 Optimizations and Observations

In this section we focus on optimizing the Multi-input receiver algorithm to achieve faster execution time. The first step was to allow compiler to optimize the code to minimize the time taken to execute the program. The second step was to optimize the algorithm by using highly optimized software libraries and look up tables. The third step was to optimize the code using various techniques like software pipeline, loop unrolling, improve cache performance and more. In order to achieve high optimization, it is important to understand the hardware architecture of the target platform. The architecture features of 674x DSP were explained in Chapter 2. The most important features for optimization are the *CPU architecture, hardware pipelining* and *memory architecture*. In this section we emphasize on various experiments performed on optimization techniques. The performance is measured in terms of time taken to extract (blind beamform) and demodulate signals in a packet slot.

4.4.1 Compiler optimization

Compiler can be configured to minimize, maximize or adapt certain attributes the program like execution speed, memory usage and power consumption. Compiler optimizations are commonly used in DSP to reduce the execution time of a program.

We know that the 674x DSP has eight functional units which can execute up to eight instruction is parallel. The basic task of the compiler optimization is to facilitate execution of more instruction in parallel. For example the following unoptimized assembly code represents the basic sum-of-products equation, $\sum_{x=0}^{N} a[N] * b[N]$.

```
        MOV A0, 0
        MOV B0, N

[loop] LOAD .D1 *A1, A6
        NOP 4
        LOAD .D2 *B1, B6
        NOP 4
        MUL .M1 A6, B6, A8
        NOP
        ADD .L1 A8, A0
        SUB .L2 B0, 1, B0
 [B0] BRANCH .S1 loop
        NOP 5
```

The location of input arrays *a* and *b* are given by the pointers *A1* and *B1*. The loop counter is initialized to the maximum value *N* in register *B0* and decremented by one in each iterations. At the end of each iteration the functional units *.S* performs the conditional branching based on the value in register *B0*. The *NOP* is the *no operation* instruction and is added to compensate for the delay associated intrusions. *NOP x* represents *x* delay cycles. The multiplication instruction has a delay of one (1), load instruction has a delay of four (4) and the branch instruction has a delay of five (5) CPU clock cycles. Other instructions can be executed without any delay. The memory pointers *A0* and *A1* are auto incremented.

We could observe that each iteration of the loop consumes 20 cycles. This can be optimized to improve the execution speed as multiple functional units can be used in parallel and the delay cycles can be reduced. The functional units can be used effectively by executing the instructions which does not depend on each other in parallel and re arranging the order of execution (out-of-order execution). The optimized sum-of-product equation can be pipelined as follows,

```
        MOV A0, 0
        MOV B0, N
```

	O0	O2	O3
Beamforming	208 ms	152 ms	145 ms
Demodulator	459 ms	335 ms	323 ms
Total	667 ms	487 ms	468 ms

Table 4.3: The compiler optimization results for the receiver application.

```
[loop] LOAD .D1 *A1, A6
 ||      LOAD .D2 *B1, B6
        SUB .L2 B0, 1, B0
 [B0] BRANCH .S1 loop
        NOP 2
        MUL .M1 A6, B6, A8
        NOP
        ADD .L1 A8, A0
```

Here the two load instruction which are independent can be executed in parallel. The '||' sign indicates that instructions are executed at the same CPU cycle. While waiting for the load instructions to complete the delay slots can be used to execute other instructions which are not depended on the outcome of the load instruction. Thus the decrement of the loop counter and the branch instruction are executed in the next two CPU cycles. Now the multiplication unit has to wait for another two CPU cycles for the load to be completed. It multiplies the two values and the results are available after one delay cycle. It is then added and saved to the register. By the time the add instruction is executed the five delay cycles associated with branch institution are completed. Now we can observe that each iteration is completed in 8 cycles instead of 20 cycles.

Apart from optimizing the hardware pipelining compiler apply various methods like loop optimizations, locality of reference (cache), machine code optimization, replacing complex instructions by simple instructions and more. For example the two cycle multiplication operation of a value by a power of two (2) can be replaced by a single cycle left-shift operation. Compiler optimizations can improve the execution speed significantly. The optimization level option for the compiler was set to -O2 or -O3. The *Debug (-g)* information was removed from the generated code. The 'opt-level=O3' provides the highest compiler optimization. The execution times of the Multi-input receiver algorithm for different compiler optimization settings are given in Table 4.3. The compiler optimized code process each packet slot in 468 ms instead of unoptimized (-O0) code in 667 ms.

4.4.2 Algorithm Logic Optimization

The Multi-input receiver algorithm was further optimized by using assembly optimized library routines, custom look-up tables and fine tuning the code structure by rearranging the statements and expressions.

	Time per slot
Beamforming	121.4 ms
Demodulator	267.3 ms
Total	388.3 ms

Table 4.4: The performance improvement after applying algorithm optimization on compiler optimized code (-O3)

TMS320C67x DSP Library (DSPLIB) [9] and TMS320C67x Fast Run-Time Support Library (FastRTS) [2] were used to provide assembly optimized floating point functions. For example *DSPF_sp_fir_cplx* function from DSPLIB was used to perform *Complex FIR Filter* and *sine, cosine, exp, atan* and more floating point math functions were used from the FastRTS library.

The *sine* and *cosine* values are used extensively by the demodulator function. The single precision sine (73 cycles per calculation) and cosine (76 cycles per calculation) [10] functions are computationally expensive. Since the used values are periodic, repetitive and chosen from a finite set of values, we have replaced them with *Lookup-Tables*. These Lookup-Tables are pre-calculated and stored in the memory during the initialization of the application. These values can be reused during the execution of the program. Hence *sine and cosine* function calls were improved to an average speed of 4 cycles per call. The time taken to process each packet slot of the Multi-input receiver after applying algorithm optimizations are shown in Table 4.4.

4.4.3 Optimization Techniques and Analysis

The algorithm optimized code needed to be further improved to achieve the required execution speed. It is observed from the profiling information that the Multi-input receiver algorithm consumes most of its time executing the loops. In order to improve the execution speed, the loops needs to run as fast as possible. A variety of techniques like software pipelining, loop unrolling, loop inversion, loop invariant code motion, improved locality of reference by using cache techniques can be applied to improve the speed. In this section we perform various experiments using these techniques and provide our observations. The performance is measured in terms of **speed up**. It is defined as the ratio of execution time for unoptimized and optimized code. The complier and algorithm optimized code (388 ms per slot) is used as the reference code. Various optimization experiments are performed on this reference code and the time taken per slot is measured. The ratio of the *reference time per slot* and the *measured time per slot* gives the speed up. The experiments are performed for realistic inputs with noise and interferences.

4.4.3.1 Experiment 1 - software pipelining

Software pipelining [16] is a loop optimization method using out-of-order execution of different iterations of the loop. It is used in processors with multiple functional units like in DSPs. The reordering of instructions is performed by the compiler.

Let us consider the following example.

```
for(i=0; i <10; i++){
     X = A[i];
     X = X+1;
     A[i] = x;
}
```

The assembly instructions for single iteration is given as,

```
LOAD A[i], A15
NOP 4
ADD A15,1
LOAD A15, A[i]
```

Here the add instruction has to wait for four (4) cycles as the value of X depends on the load instruction. However there are no dependencies between loading of *A[i]* of the second iteration and the first iteration of the loop. Hence it may start before the add instruction of the first iteration begins. Software pipelining tries to keep the functional units busy by executing instructions from different iterations of the loop which are not dependent on each other. This scenario is depicted in figure 4.3. In this example, the non-software pipelined loop would consume 49 cycles for 7 iterations where as the software pipelined loop executes it in 13 cycles. The region where maximum or all the functional units are busy is called *loop kernel*. The stages above the kernel are called *prolog* and below are called *epilog*. Software pipelining is effective when different iteration of the loop are independent.

LOAD A[0]						
NOP	LOAD A[1]					
NOP	NOP	LOAD A[2]				
NOP	NOP	NOP	LOAD A[3]			
NOP	NOP	NOP	NOP	LOAD A[4]		
ADD A10, 1	NOP	NOP	NOP	NOP	LOAD A[5]	
LOAD A10, A[0]	ADD A11, 1	NOP	NOP	NOP	NOP	LOAD A[6]
	LOAD A11, A[1]	ADD A12, 1	NOP	NOP	NOP	NOP
		LOAD A12, A[2]	ADD A13, 1	NOP	NOP	NOP
			LOAD A13, A[3]	ADD A14, 1	NOP	NOP
				LOAD A14, A4]	ADD A15, 1	NOP
					LOAD A15, A[5]	ADD A16, 1
						LOAD A10, A[6]

Figure 4.3: Software pipelining of loops. Multiple iterations of the loops are executed in parallel

Software pipelining can be enabled in the CCS Compiler at optimization level O2 and O3. In TI 674x DSP platform it is supported by hardware SPLOOP buffers, which helps to reduce the code size. The compiler can generate information about the software

pipelining applied on loops. Figure 4.4 shows an example of the compiler generated software pipelined loop information. This information is very useful to understand and improve the performance of loops. Examples of information generated by the compiler are:

```
*----------------------------------------------------------------------------*

;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Known Minimum Trip Count         : 2
;*      Known Maximum Trip Count         : 2
;*      Known Max Trip Count Factor      : 2
;*      Loop Carried Dependency Bound(^) : 4
;*      Unpartitioned Resource Bound     : 4
;*      Partitioned Resource Bound(*)    : 5
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     2        3
;*      .S units                     4        4
;*      .D units                     1        0
;*      .M units                     0        0
;*      .X cross paths               1        3
;*      .T address paths             1        0
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           0        1     (.L or .S unit)
;*      Addition ops (.LSD)          6        3     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             3        4
;*      Bound(.L .S .D .LS .LSD)     5*       4
;*

;*      Searching for software pipeline schedule at ...
;*         ii = 5  Register is live too long
;*         ii = 6  Did not find schedule
;*         ii = 7  Schedule found with 3 iterations in parallel
;*      done
;*
;*      Epilog not entirely removed
;*      Collapsed epilog stages     : 1
;*
;*      Prolog not removed
;*      Collapsed prolog stages     : 0
;*
;*      Minimum required memory pad : 2 bytes
;*
;*      Minimum safe trip count     : 2
;*
;*----------------------------------------------------------------------------*
```

Figure 4.4: Example of software pipeline loop information generated by the compiler. Useful for further optimization.

- Minimum and maximum number of the times the loop was executed and hence the number of times the loop can be unrolled.
- The number of times each functional unit, register files and cross paths accessed. Cross path is the connection between set of functional units (Data path 1 and Data path 2) in the TI 6x DSP core. This information can be used to balance the usage of the functional units.
- Register usage.

In our first experiment the effect of **software pipelining** is studied on Multi-input receiver algorithm. We have enabled the software pipelining for optimization level O3.

#packets/slot	#demodulated packet	Unoptimized	Optimized	Speed up
1	1	382.44 ms	372.41 ms	1.029
2	2	382.73 ms	372.54 ms	1.029
4	4	382.89 ms	372.16 ms	1.029
More than 4	4	382.56 ms	372.45 ms	1.029

Table 4.5: Software pipelining

The results of this experiment are given in Table 4.5. Software pipelining is often used in combination with *loop unrolling*. This combination of techniques often provides far better optimization than software pipelining or loop unrolling alone.

4.4.3.2 Experiment 2 - Memory Aliasing

Memory aliasing is the situation where same memory locations are accessed using different symbolic names in the program. Thus modifying the data through one symbol modifies all the values associated with every alias of the data, which may produce unexpected results. A very common scenario of symbolic names in programming are memory pointers. Memory pointers (for example arrays) are typically used in function calls as arguments. Let us consider a function as shown bellow,

```
void function(int * a, int * b, int * out){
    for(i=0; i < N; i++){
        out[i] = a[i] + b[i];
    }
}
```

In this example aliasing occurs if *a,b* or *out* point to the same memory location at any stage of the execution. If the compiler does not know explicitly that the two pointers do not overlap, it expects that aliasing may occur. Hence it does not perform any optimization on the code in order to avoid incorrect results. This scenario can occur in Code Composer Studio tool if the caller function is located in different file. The assembly code for the function is given as,

```
LOAD .D1 a,A0
NOP 4
LOAD .D2 b, B0
NOP 4
ADD A0, B0, A3
STORE .D1 A3, out
```

If it can be told to the compiler that the symbols (pointers) don't have dependencies, compiler can be aggressive in optimizing the code by pipelining the instructions efficiently. Memory aliasing can be avoided by using the *restrict* keyword. Restrict tells the compiler that the symbol (pointer) is the only access point to the object (memory). The usage of *restrict* keyword is given bellow,

#packets/slot	#demodulated packet	Unoptimized	Optimized	Speed up
1	1	382.12 ms	367.82 ms	1.041
2	2	382.23 ms	368.12 ms	1.041
4	4	382.34 ms	367.92 ms	1.041
More than 4	4	382.45 ms	368.11 ms	1.041

Table 4.6: Memory Aliasing

void function(int * *restrict* a, int * *restrict* b, int * out)

In the above example since the compiler knows that the two inputs are not alias to each other it can execute them in parallel as,

```
      LOAD .D1 a,A0
||    LOAD .D2 b, B0
      NOP 4
      ADD A0, B0, A3
      STORE .D1 A3, out
```

In this experiment the effect of **Memory Aliasing** on Multi-input receiver algorithm is studied. The results of this experiment are given in Table 4.6. It is observed that the performance gained only from eliminating memory aliasing is very marginal.

4.4.3.3 Experiment 3 - Loop invariant code motion

The loop invariant code motion is a technique in which the invariant statements or expressions are moved outside the body of the loop without affecting the results of the program. Hence the invariant code is executed less often providing speedup.

Let us consider the example,

```
for(i=0; i < N; i++){
   for(j=0; j < M; j++){
      a = x + y;
      b = j * i + x * i;
      c = j * j + y * x;
   }
}
```

In this example the variables a (x+y) and y*x can be calculated outside both the loops. The variable x*i can be calculated outside the inner loop as shown,

```
a = x + y;
n = y * x;
```

#packets/slot	#demodulated packet	Unoptimized	Optimized	Speed up
1	1	382.12 ms	364.23 ms	1.049
2	2	382.23 ms	364.21 ms	1.049
4	4	382.34 ms	364.12 ms	1.049
More than 4	4	382.45 ms	364.32 ms	1.049

Table 4.7: Loop Invariant Code Motion

```
for(i=0; i < N; i++){
   m = x * i;
   for(j=0; j < M; j++){
      b = j * i + m;
      c = j * j + n;
   }
}
```

The advantage of loop invariant code motion is that the invariant values can be calculated and stored in internal registers. It eliminates calculating the values in each iterations and saves valuable memory access cycles. This is one of the tricks applied to speed up the loops in the *demodulator* function. The inner loop in demodulator function has invariant expressions which involves many load operations. For 674x DSP architecture load operation is expensive as it consume five (one execution and four delay) CPU cycles. In this experiment the effect of **Loop Invariant Code Motion** is studied. The results of this experiment are given in Table 4.7. The invariant Code Motion Technique provides better speed up compared to the techniques discussed above. The disadvantage of this techniques is that it can affect the readability of the code.

4.4.3.4 Experiment 4 - Loop Unrolling

The Loop unrolling is a loop transformation technique to optimize the execution speed of a code. It provides speed up by reducing or eliminating the 'end of loop' test in each iteration by duplicating the body of the loop. The price one has to pay for loop unrolling is that of increased code size.

let us consider the sum-of-products example given bellow,

```
        MOV A0, 0
        MOV B0, N

[loop] LOAD .D1 *A1, A6
 ||     LOAD .D2 *B1, B6
        SUB .L2 B0, 1, B0
 [B0] BRANCH .S1 loop
        NOP 2
```

```
        MUL .M1 A6, B6, A8
        NOP
        ADD .L1 A8, A0
```

In this example the loop needs to be iterated N times. The loop can be unrolled by a factor 2 by replicating the code inside the loop as,

```
        MOV A0, 0
        MOV B0, N

[loop] LOAD .D1 *A1, A6
 ||     LOAD .D2 *B1, B6
        NOP 2
        MUL .M1 A6, B6, A8
        NOP
        ADD .L1 A8, A0

        LOAD .D1 *A1, A6
 ||     LOAD .D2 *B1, B6
        SUB .L2 B0, 2, B0
 [B0] BRANCH .S1 loop
        NOP 2
        MUL .M1 A6, B6, A8
        NOP
        ADD .L1 A8, A0
```

We could see that the branch instructions required by the loop is reduce by the unroll factor. This code can execute faster as the branch instruction are expensive on any platform. In TI 674x DSP branch instruction requires 6 cycles to execute. This code can be further optimized by using out-of-order execution techniques as shown,

```
        MOV A0, 0
        MOV B0, N

[loop] LOAD .D1 *A1, A6
 ||     LOAD .D2 *B1, B6
        NOP 2
        MUL .M1 A6, B6, A8
 ||     LOAD .D1 *A1, A7
 ||     LOAD .D2 *B1, B7
        SUB .L2 B0, 2, B0
        ADD .L1 A8, A0
 ||[B0] BRANCH .S1 loop
```

#packets/slot	#demodulated packet	Unoptimized	Optimized	Speed up
1	1	382.44 ms	348.296 ms	1.102
2	2	382.84 ms	348.298 ms	1.102
4	4	382.42 ms	348.291 ms	1.102
More than 4	4	384.67 ms	348.299 ms	1.102

Table 4.8: Loop unrolling

#packets/slot	#demodulated packet	Unoptimized	Optimized	Speed up
1	1	382.44 ms	324.763 ms	1.172
2	2	382.84 ms	324.759 ms	1.172
4	4	382.42 ms	324.761 ms	1.172
More than 4	4	384.67 ms	324.760 ms	1.172

Table 4.9: Software pipelining applied on loop unrolled code

```
NOP 2
MUL .M1 A7, B7, A8
NOP
ADD .L1 A8, A0
```

The loop execution speed by can be improved by choosing a higher unroll factor. In practice unrolling speeds up the code to a certain point after which only the code size increases, not speed. The loop unrolled code can be further improved by using out-of-order execution techniques.

The advantage of loop unrolling is that it can provide additional performance gain by using software pipelining. The disadvantage is the increased code size which is undesirable for embedded applications. Increase in code size can potentially increase the *cache misses* which may affect the performance. Normally a trade off is made among unrolling factor, code size and performance. If loop unrolling is done manually it can make the code unreadable.

In this experiment the effect of **Loop Unrolling** is studied. The results of this experiment are given in Table 4.8. As expected the *loop unrolling* provides significant speed up. This technique is able to optimize the loops in demodulator as well as beamforming functions. We have also tested the *loop unrolling* along with *software pipelining* technique. We could observe that the loop unrolling is more effective when used with software pipelining and vica versa. We had to be careful in choosing the unroll factor so that software pipeline does not collapse to find a schedule. The results are shown in Table 4.9.

#packets/	#demodulated packet	Unoptimized	Optimized	Speed up
1	1	382.12 ms	376.23 ms	1.015
2	2	382.23 ms	376.85 ms	1.015
4	4	382.94 ms	376.64 ms	1.015
More than 4	4	382.83 ms	376.54 ms	1.015

Table 4.10: Data Alignment

4.4.3.5 Experiment 5 - Data Alignment

CCS tool allows the data to be aligned on any boundary on memory. It is observed that if the data values are aligned on *word (4 bytes)* or *double word (8 bytes)* boundary on the memory, the Memory Management Unit (MMU) can access them faster. The On-chip bandwidth of the memory is 256 bits (8 words) as explained in Chapter 2. This means that the 674x architecture allows up to 8 bytes to be accessed in a single load operation. Data alignment makes it possible to access multiple data types of size less than 8 bytes to be accessed in single load operation. For example instead of loading four 16-bit data types in four operations, it can be loaded in one double word (8 byte) access if aligned on 2 byte boundary.

This allows to use *word wide optimizations* and *packed data processing* techniques to achieve better performance. Let us consider four 16 bit variables a0, a1, b0 and b1. Let us assume we need the results (a0 * b0) and (a1 * b1). The normal way to perform this operation is to load a0 and b0 to registers and perform the multiplication operation and repeat the same process for a1 and b1. If the data values a0, a1 and b0, b1 are aligned on a 2 byte boundary we can load the values a0, a1 to a single register say A0 (32 bit) using a single load operation (LOAD *ptr1, A0). The same can be done for values b0 and b1 (LOAD *ptr2, A1). Word wide optimization allows multiple 8bit and 16 bit operations on a single 32 bit functional unit. In this example MPY instruction perform the lower 16 bit multiplication and MPYH perform the higher 16 bit multiplication. The advantage of this technique is that we would require less load instructions and the multiplications can be performed in parallel. Packed data processing is very similar to word wide optimization. It is also referred as single instruction multiple data (SIMD), for which each instruction has more than one data value.

For DSP applications data alignment can provide significant performance. Our receiver application requires large data buffers to store input and intermediate data. In this experiment the effect of **Data Alignment** on Multi-input receiver is studied. The results of this experiment are given in Table 4.10. It is observed that the gain obtained from Data Alignment is very marginal. The reason is the limited usage of packed data processing and word-wide optimization techniques.

4.4.3.6 Experiment 6 - Loop Inversion

Loop inversion is a technique reduce branch overhead. In Loop inversion *while* loops are replaced by *do-while with if block* or *for* loops. It is observed that the number of branch

#packets/slot	#demodulated packet	Unoptimized	Optimized	Speed up
1	1	382.92 ms	374.94 ms	1.021
2	2	382.95 ms	374.05 ms	1.021
4	4	382.21 ms	374.33 ms	1.021
More than 4	4	382.54 ms	374.21 ms	1.021

Table 4.11: Loop Inversion

instructions is reduced when the loop is converted. Branch miss can affect the hardware pipeline and cause delay. Loop inversion can reduce the branch instructions and thus provide speed up.

In this experiment the effect of **Loop inversion** is studied. The results of this experiment are given in Table 4.11. The Loop inversion technique is effectively used in the main demodulator loop. It is observed that this has an impact on the code speed.

4.4.3.7 Experiment 7 - Memory management

Using memory efficiently is important in optimizing the performance. In Chapter 2 we have described the memory architecture of TI 674x DSP platform. It has two level cache architecture, L1 and L2. Level L1 has 32 KBs of separate instruction and data cache. L2 is 256 KB shared instruction and data cache. L2 is an addressable memory unlike L1. L2 can be used as a mapped memory, cache or a combination of both. Additionally there is 128 KB of on chip DRAM memory and external memory (up to 4 GB) which can be accessed through external memory interface.

```
-stack           0x0001FFFF
-heap            0x00016200

MEMORY
{
   dsp_l2_ram:      ORIGIN = 0x11800000  LENGTH = 0x00040000
   shared_ram:      ORIGIN = 0x80000000  LENGTH = 0x00020000
   external_ram:    ORIGIN = 0xC0000000  LENGTH = 0x08000000
   arm_local_ram:   ORIGIN = 0xFFFF0000  LENGTH = 0x00002000
}

SECTIONS
{
   .text   : ALIGN(8) {}   > external_ram
   .const  : ALIGN(8) {}   > external_ram
   .bss    : ALIGN(8) {}   > external_ram
   .far    : ALIGN(8) {}   > external_ram
   .switch : ALIGN(8) {}   > external_ram
   .stack  : ALIGN(8) {}   > external_ram
   .data   : ALIGN(8) {}   > external_ram
   .cinit  : ALIGN(8) {}   > external_ram
   .sysmem : ALIGN(8) {}   > external_ram
   .cio    : ALIGN(8) {}   > external_ram
   .recv   : ALIGN(8) {}   > external_ram
}
```

Figure 4.5: Example of Linker Command File.

Using the linker command file of the CCS tool, various linked objects can be allocated to specific location on memory on the target platform. The **MEMORY** specifies the regions on the memory and the **SECTION** specifies how the linker objects are associated with memory regions. The '.text' section contains the code and '.bss' contain the global and static variables. The linker command file used to generate the primary results is shown in Figure 4.5. The linker command file plays a key role in memory management and hence providing better performance on the target platform.

To optimize performance it is important that all memory requirements of the application are satisfied by on-chip memory. Additionally from the date sheet [1] external memory access causes more power consumption than internal memory access. When it is not possible to keep everything on-chip, we have to look for ways to keep the critical code and data in the on-chip memory to provide faster execution. This can be achieved by defining custom sections in the code/data and allocate them in the on-chip memory using the linker command file.

It is important to allocate *stack* and *heap* sections on-chip to improve the performance. The stack is a location in the memory where local variables are stored when a function is called. The stack location is shared for all the functions. Heap section is used for dynamic memory is allocation. .

Cache Memory

```
-stack           0x00021000
-heap            0x00016500

MEMORY
{
   dsp_l2_ram:      ORIGIN = 0x11800000  LENGTH = 0x00040000
   shared_ram:      ORIGIN = 0x80000000  LENGTH = 0x00020000
   external_ram:    ORIGIN = 0xC0000000  LENGTH = 0x08000000
   arm_local_ram:   ORIGIN = 0xFFFF0000  LENGTH = 0x00002000
}

SECTIONS
{
   .text   : ALIGN(8) {}   > shared_ram
   .const  : ALIGN(8) {}   > dsp_l2_ram
   .bss    : ALIGN(8) {}   > dsp_l2_ram
   .far    : ALIGN(8) {}   > external_ram
   .switch : ALIGN(8) {}   > dsp_l2_ram
   .stack  : ALIGN(8) {}   > dsp_l2_ram
   .data   : ALIGN(8) {}   > dsp_l2_ram
   .cinit  : ALIGN(8) {}   > external_ram
   .sysmem : ALIGN(8) {}   > dsp_l2_ram
   .cio    : ALIGN(8) {}   > dsp_l2_ram
   .cri1   : ALIGN(8) {}   > dsp_l2_ram
   .cri2   : ALIGN(8) {}   > dsp_l2_ram
   .cri3   : ALIGN(8) {}   > dsp_l2_ram
   .cri4   : ALIGN(8) {}   > dsp_l2_ram
}
```

Figure 4.6: Linker command file used for speed optimization

Cache memory keeps a local scratch copy of code and data in L1 memory. L1 Cache memory is the closest to the processor and has lowest latency. Cache memory provides

#packets/slot	#demodulated packet	Unoptimized	Optimized	Speed up
1	1	382.44 ms	42.937 ms	8.901
2	2	382.84 ms	42.963 ms	8.901
4	4	382.42 ms	42.534 ms	8.901
More than 4	4	384.67 ms	42.971 ms	8.901

Table 4.12: Cache Memory & Efficient Memory management

#packets/slot	#demodulated packet	Unoptimized	Optimized	Speed up
1	1	382.44 ms	24.492 ms	15.614
2	2	382.84 ms	24.501 ms	15.614
4	4	382.42 ms	24.499 ms	15.614
More than 4	4	384.67 ms	24.498 ms	15.614

Table 4.13: Combined Optimization

performance by reusing the local copies of instruction and data in cache memory. Cache misses are costly as it would require fetching fresh instruction and data from higher level memories which have larger latencies. L1 program is a direct mapped cache [13] where as L1 data is 2 way set associative cache [14]. L2 can be used as 4 way set associative cache or can be used as RAM memory. To improve performance it is important to avoid L1 instruction cache misses. Since the L1 program cache is direct mapped, proper placement of code section in the memory is needed. It can be achieved by good code placement strategies.

In this experiment the effect of **Cache Memory & Efficient Memory management** is studied. The results of this experiment are given in Table 4.12. Based on the profile information generated by the compiler we could place the critical sections of the code in memory close to the processor. The L2 cache is effectively used as an on-chip memory for that purpose. The linker command file used for achieving the results is shown in Figure 4.6. We could observe that memory management and reducing instruction cache miss by proper placement of code in the appropriate memory can provide the best performance gain among all the techniques studied.

4.4.3.8 Experiment 8 - All Optimizations combined

In this experiment the combined effect of **all the optimization** techniques are studied. The results of this experiment are given in Table 4.13.

We observed that when all the optimization techniques were applied together the multi-input receiver could perform the beamforming and demodulation in less than 25 ms. This satisfies the real-time speed up criteria of the multi-input receiver. This has a speed up factor of over 15.6 to the compiler optimized code. We could observe that the effect of various optimization techniques depends on each other. The performance

Power Consumption	Idle Power	Active Power
Measured Board Power	650mW	1150mW
SATA controller	330mW	330mW
Peripherals Power	40mW	40mW
External Memory	200mW	200mW
Other Controllers	35mW	35mW
Power consumed DSP (CPU and Internal Memory)	45mW	445mW

Table 4.14: Power consumption

depends on how well the instructions can be pipelined to keep the functional units busy and reducing the latencies by modifying the instructions. For example a software pipelining is more effective on an unrolled loop than applied alone. Hence it was possible to achieve more performance than the contribution of individual techniques by making proper trade offs.

4.4.3.9 Experiment 9 - Power Consumption Analysis

In this experiment we study the **power** consumption of the DSP processor. We have measured the power consumption of the target board 'ZOOM OMAP-L138 EXPERIMENTER KIT' [11]. Though the silicon used in the experiment board is a preliminary silicon and is not meant to be power accurate it could give comparable results. The only way to measure the power consumption on the DSP was by measuring the voltage drop across a known resistor connected in series. It is done by measuring the current at the output of a DC-DC converter using a multimeter. The register is located between the DC-DC converter and the DSP core. The measurements are given in Table 4.14. We have compared the calculated power with the values on the datasheet for various operating modes of the DSP. From the datasheet the idle power is 11 mW and the power on active mode is 475 mW. We observed that the measure power was not accurate and had an error margin of +20%.

The measured power is the accumulative power consumed by the DSP, peripherals, external memory and other controllers (SATA clock generator, Ethernet, USB etc) on board. Hence the DSP power (mainly the processing units and internal memory) consumption is calculated by subtracting the other power consumption which is available from the datasheets [7].

From the datasheet the DC-DC converter has an efficiency of 0.8. So to calculate the actual power consumption we have to compensate for this value. Hence the power is 125% (1/0.8) of the measured value. The **Idle power** is 45mW x 1.25 = 56mW and the **Active power** is 445mW x 1.25 = 556mW. These values fit within the power requirement (600 mW) of the receiver system. Since the measured values depends on the silicon and the inaccuracy of the measurements we assume that the actual values are smaller than observed.

4.4.3.10 Analysis

In this section we study the effect of different optimization techniques applied to Multi-input receiver application.

Figure 4.7 gives the comparison of the Multi-input receiver speed performance during various stages of optimization on the target platform. The test inputs with noise and interferences were generated using a simulation tool. The performance is measured in teams of time consumed (in millisecond) to separate out and demodulate each group of overlapping signals (slots). We compare the performance of unoptimized code with LINPACK library integrated, compiler optimized, algorithm logic optimized and finally the Optimized code. The speed performance of the Multi-input receiver system has improved from 2.67 seconds per slot to 24.5 ms per slot. From experiment 9, the power consumption of the Multi-input receiver on the target platform is within the power budget of the system in its active mode.

The comparison of *Speedup* provided by each optimization technique is given in Figure 4.8. It is obvious that among the different optimization techniques applied on the embedded DSP platform, cache and memory management has most impact on performance.

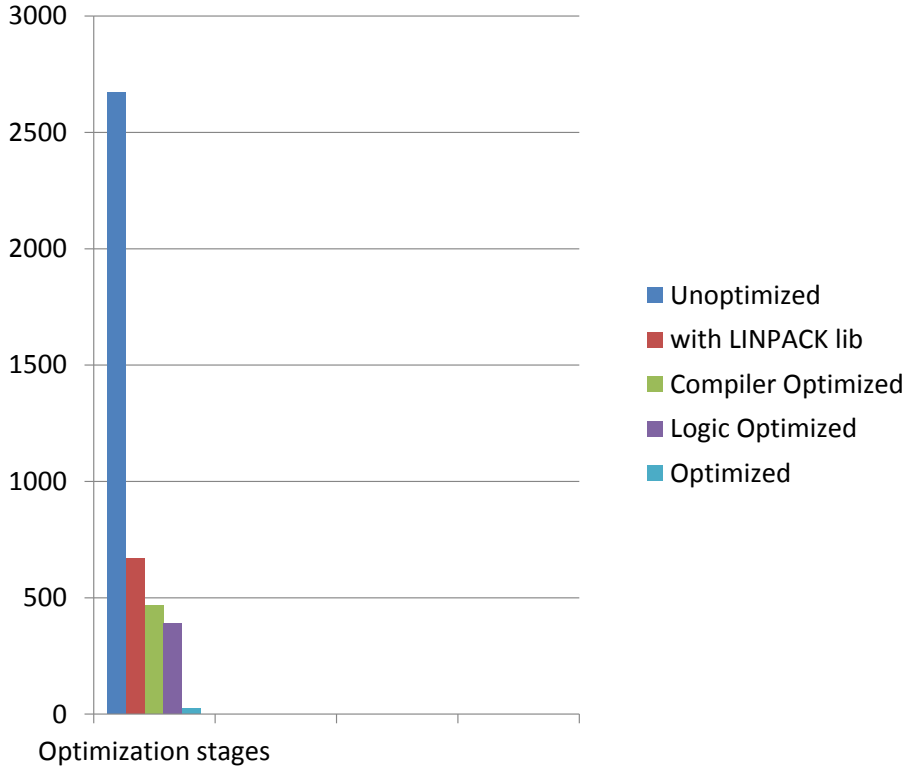

Figure 4.7: Comparison of the various stages of Optimization for Multi-input receiver (time consumed per millisecond).

#packets/slot	Total	beamforming	demodulator
1	14.96 ms	7.12 ms	3.720 ms
2	18.12 ms	7.12 ms	7.641 ms
4	24.49 ms	7.12 ms	14.37 ms
More than 4	24.49 ms	7.12 ms	14.38 ms

Table 4.15: Performance in the absence of noise for optimized code

It is observed that the *beamforming* consumes approximately 7 ms and the *demodulator* consumes 14 ms per slot in the final optimized version. The input signals with less number of packets per slot are expected perform faster than input signals with four or more packets per slot. In practice the time consumed for different number of packets per slots are almost the same. The reason for such behavior is false signal detection. To test the Multi-input receiver for real scenarios the test inputs are considered with noise such that the 'signal to noise ratio' is kept low. In the presence of noise, false signals are detected which invokes the demodulator more number of times than actual number of signals present in input. In the current implementation the demodulator can be invoked maximum of four times per slot. We have provided the speed performance of the Multi-input receiver for test inputs without noise. The observations are provided in Table 4.15. We could observe that the demodulator shows better performance in the absence of noise.

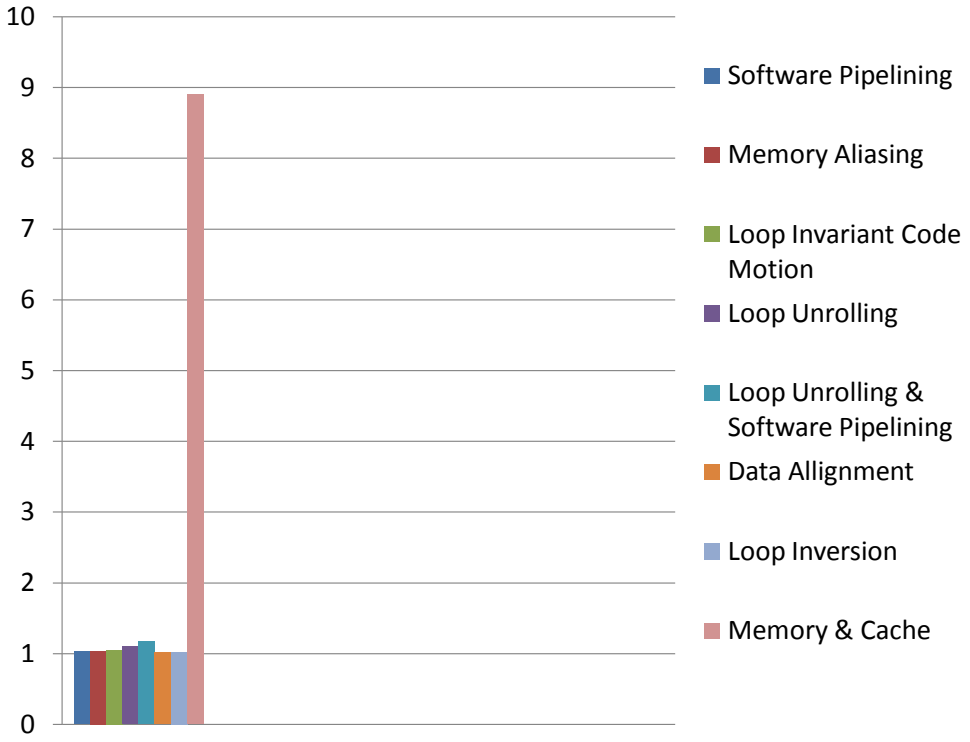

Figure 4.8: Comparison of the effect of various optimization techniques (speedup).

We could observe from the memory map file generated by CCS tool that the code and data memory requirements are fulfilled by the on chip SRAM and addressable L2 memory used as SRAM. As a result no external memory is required by the algorithm.

4.5 Summary

In this chapter we have described the various optimization techniques applied to achieve real-time performance. Among the different optimization techniques applied on the embedded DSP platform, cache and memory management has the most impact on performance. By applying all of the studied methods we could achieve the required performance in terms of processing speed. The external memory power consumption is expensive from the datasheets. The on-chip memory alone is sufficient to handle the Multi-input receiver algorithm. By reducing external memory usage we could reduce the power consumption of the system. We could also observe that the power consumption of the target platform is within the power budget of the system. We could achieve the goals without compromising on message recovery or demodulation.

5 Conclusion And Future Scope

5.1 Conclusion

When signals are transmitted from distinct sources at the same time and at the same frequency they get overlapped with each other. It is called co-channel interference. Constant Modulus Algorithms (CMA) is a spatial signal processing technique to negate such interferences for constant modulus signals such as phase shift keying (PSK) and frequency shift keying (FSK). It use a *beamforming* technique which is based on a antenna array to separate out individual signals. This master thesis work was inspired by the practical aspects of the 'Analytical Constant Modulus Algorithm' (ACMA) proposed by Alle-Jan van der Veen and Arogyaswami Paulraj. The ACMA algorithm proposes an efficient analytical approach to separate out individual signals using an antenna array. The ACMA algorithm is a **blind beamforming algorithm** as it does not require any knowledge of the signals and the channel.

ACMA solution to blind beamforming problem demonstrates several advantages over other Constant Modulus Algorithms (CMA). The conventional CMAs can detect only one signal from the set of overlapping signals, where ACMA can detect more signals based on the number of antennas. A multi-stage CMA can be used to detect more than one signal based on the number of antennas, but it requires more processing. Moreover in practice ACMA demonstrates better performance to CMA in detecting signals. Another advantage is that ACMA requires only a moderate amount of samples.

The scope of this thesis work was to implement a low cost, low power real-time Multi-input receiver application. The Multi-input receiver was capable of handling co-channel signal from different sources which are partially and fully overlapped. The basic requirements of the Multi-input receiver system was that each group of overlapping messages (slots) to be processed in less than **25 ms** and the power consumption to be less than **600 mW**.

The front end of the receiver system consists of an antenna array of four (4) antennas. The front end extracts the desired frequency channels, performs analog to digital conversion and down sampling of antenna array inputs. The receiver system separates out individual signals using the blind beamforming (ACMA) algorithm. The demodulator performs demodulation. It identifies a valid signal (message) by its start and end flags and by using an error detection mechanism (Cyclic Redundancy Check).

The Multi-input receiver system is software defined as the important functionalities, *beamforming* and *demodulation* are implemented in software. The platform chosen from implementation is Texas Instruments TMS320C6748 (674x) DSP. The reasons were the

following. It is a high performance, low power DSP processor, well suited for signal processing applications. It has a *load and store RISC* architecture. It can execute up to 8 instructions in parallel at 375/456 MHz frequency. It can perform both fixed point as well as floating point arithmetic operations. These features can provide sufficient computations to achieve the real-time performance required by the Multi-input receiver. The TMS320C6748 is low power device as it consumes 470 mW in the active mode. It satisfies our basic power requirement. However power is a critical resource for the Multi-input receiver on the embedded platform. Hence the DSP processor was clocked at a lower frequency of **300 MHz** to save more power. The DSP has a rich set of peripherals and high amount of on chip memory.

The blind beamforming algorithm requires certain functions such as *singular value decomposition (svd)* from the Linear Algebra Package (LAPACK) library. Since there were no free LACPCK implementations were available for Texas Instruments DSP platform an open source WINDOWS implementation *CLAPACK* library was ported to TI 674x DSP platform. The observed performance of the Multi-input receiver was 2.67 seconds per slot to perform beamforming and demodulation for up to four messages. We observed that the LAPACK functions requires significant amount of time to perform. Since the LAPACK library was not optimized it was replaced by a third party highly optimized LINPACK library. Additionally the algorithm was modified from double precision to single precision arithmetic. The observed performance was improved to 667 ms per slot. The receiver algorithm was then optimized by using the *compiler* tool chain which gave us a performance of 468 ms per slot. The Multi-input receiver algorithm was further optimized by using assembly optimized library routines from Texas Instruments such as FIR filters and *look-up tables* for sine and cosine. The observed performance was 388 ms per slot.

The code needed to be further optimized in order to achieve real-time performance (less than 25 ms per slot). Since most of the execution time of the Multi-input receiver was spend in 'loops' they were targeted for optimization. Various optimization techniques were experimented and the effects were studied. The optimization techniques involve software pipelining, loop unrolling, avoiding memory aliasing problem, providing data alignment in memory, using efficient memory management and utilizing cache memory. We observed that each technique has variable impact on the performance. Techniques such as loop unrolling is more effective when applied along with software pipelining than applied alone. We could observe that memory management and cache provided the best performance gain to the receiver application. We gained performance by proper placement of critical code in memory close to the processor so as to minimize instruction cache misses. We observed that the cache misses are expensive as it results in re-fetching of instructions from higher level memory with additional latency. Moreover minimizing the external memory access can save power.

The final optimized code has a performance of **24.5 ms** per slot which is well within the limits of the real-time requirement. We also observed that the power consumption of the system is approximately **500 mW** which is also within the power budget allocated for the system. Hence all the objectives of the thesis work were achieved.

5.2 Future Work

In this work we have implemented a low power, low cost Multi-input receiver using blind beamforming algorithm. This algorithm can find many applications especially for low power mobile devices which require wireless communication. In the current implementation we have used only 4 antennas. The ACMA algorithm is scalable which means more signals can be detected by increasing the number of antennas and proving sufficient computations. It is observed that the processing load increases exponentially with the number of antennas. It will be interesting to study the scalability of the Multi-input receiver and the impact on processing load. Another scope of this work is to further optimize the beam forming and demodulation algorithms. We can optimize the demodulator function using fixed point arithmetic. Further optimization can be done by storing the antenna data in 16 bit format instead of 32 bit and utilizing the packed data processing and word wide optimization techniques provided by Texas Instruments DSP processor. 16 bit data storage can save the memory to store the data values.

Bibliography

[1] *C6747/45/43 power consumption summary*, Texas Instruments.

[2] *C67x fast run-time support (rts) library*, Texas Instruments.

[3] *Code composer studio (CCStudio) integrated development environment (IDE) v4.x - CCSTUDIO - TI tool folder*, Texas Instruments.

[4] *Gdd7000 - hand coded optimised linpack library*, Kane Computing, U.K.

[5] *Lapack: linear algebra package*, Netlib.

[6] *Omap-l138 c6-integratm dsp+arm processor*, Texas Instruments.

[7] *Omap-l138 power consumption summary*, Texas Instruments.

[8] *Optimizing software in c++*, Agner Fog.

[9] *Tms320c67x dsp library*, Texas Instruments.

[10] *Tms320c67x fastrts library user's guide (rev. a)*, Texas Instruments.

[11] *Zoom omap-l138 experimenter kit*, Logic PD.

[12] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK users' guide*, third ed., 1999.

[13] UMD CS, *Direct mapped cache*, `http://www.cs.umd.edu/class/spring2003/cmsc311/Notes/Memory/direct.html`, 2003.

[14] ———, *Set associative cache*, `http://www.cs.umd.edu/class/spring2003/cmsc311/Notes/Memory/set.html`, 2003.

[15] Paul M. Embree, *C language algorithms for real-time dsp*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.

[16] R.B. Jones and V.H. Allan, *Software pipelining: a comparison and improvement*, Microprogramming and Microarchitecture. Micro 23. Proceedings of the 23rd Annual Workshop and Symposium., Workshop on, nov 1990, pp. 46 –56.

[17] N. Seshan, *High velociti processing [texas instruments vliw dsp architecture]*, Signal Processing Magazine, IEEE **15** (1998), no. 2, 86 –101, 117.

[18] S.G. Smith, R.W. Morgan, and J.G. Payne, *Generic asic architecture for digital signal processing*, oct. 1989, pp. 82 –85.

[19] Steven W. Smith, *The scientist and engineer's guide to digital signal processing*, California Technical Publishing, San Diego, CA, USA, 1997.

[20] J. Treichler and B. Agee, *A new approach to multipath correction of constant modulus signals*, Acoustics, Speech and Signal Processing, IEEE Transactions on **31** (1983), no. 2, 459 – 472.

[21] TMS320C6000 Code Composer Studio Tutorial, 2010, Texas Instruments.

[22] G. Ungerboeck, D. Maiwald, H.-P. Kaeser, P. R. Chevillat, and J. P. Beraud, *Architecture of a digital signal processor*, IBM Journal of Research and Development **29** (1985), no. 2, 132 –139.

[23] A.-J. van der Veen and A. Paulraj, *An analytical constant modulus algorithm*, Signal Processing, IEEE Transactions on **44** (1996), no. 5, 1136 –1155.

[24] www.ti.com, *Tms320c6748 fixed/floating-point dsp*, `http://www.ti.com`, 2009, [SPRS590C].

[25] ———, *OMAP-L138 Low-Power Applications Processor*, `http://focus.ti.com.cn/cn/lit/ds/sprs586c/sprs586c.pdf`, 2010, [SPRS586B].