Implementing IS-95, the CDMA Standard,

on TMS320C6201 DSP

by

Xiaozhen Zhang

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical Engineering

and Master of Engineering in Electrical Engineering

at the Massachusetts Institute of Technology

May 21, 1999

[June 1999]

Author_____
Department of Electrical Engineering and Computer Science
May 7, 1999

Certified by_____
Victor Michael Bove
Thesis Supervisor

Accepted by_____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

ACKNOWLEDGEMENTS

I wish to express my appreciation and sincere thanks to a number of people for their help during my time at Texas Instruments (TI) Wireless R&D Lab.

I would like to thank my supervisor, Dr. Alan Gatherer, who not only gave me the opportunity to write a thesis, but made sure that I had all the resources I needed to do so. And for that, I also need to thank Dr. Edward Esposito for helping me to get the job with Alan. I am especially grateful to my mentor, Dr. Aris Papasakellariou, in teaching me every bit of knowledge about CDMA and IS-95 that I had no background whatsoever previously.

I also owe much thanks to other members of my group. Mr. Dale Hocevar, with his expertise in DSP implementation, gave me many useful suggestions for my work. Dr. Anand Dabak became my acting-mentor for the time when Aris was working on projects with another group.

The folks in the neighboring Wireline R&D Lab also deserve a great deal of my appreciation. Mr. Yaser Ibrahim was my help wizard for using the GO DSP software. Mr. Dennis Mannering and Dr. Nirmal Warke were always present when needed in answering other C6x related questions or software questions that I had.

I was also extremely fortunate in knowing the people who actually helped to build the C6x compiler: Mr. David Bartley and Mr. Paul Fuqua. Without David's help, I can not imagine how much trouble I would have to go through to get the Viterbi code to become so efficient as in its current form.

Mr. Partha Mukherjee and Mr. Ching-Yu Hung are members of other branches who I had the opportunity to meet and gave me guidance on both the IS-95 standard and C6x implementation.

My office was located within the Control R&D group. The folks there are just as nice to me as they possibly can. Especially my friendly neighbor, Dr. Steve Fedigan, gave much help on programming in C and other questions I liked to pump up to him. Mr. Tod Wolf and Dr. Tim Schmidl along with Steve were my lunch partners. Their inspiring discussions at lunchtime often brought lots of joy into my workday life.

My most sincere thanks go to my thesis advisor, Dr. Mike Bove of MIT Media Lab. Dr. Bove's encouragement and support made my time away from MIT worry-free. It was really assuring to know that he was one person that I could really count on.

Last, but not least, my thanks go to my family, my mother, father, and my sister. Their love and support are always so strong throughout my life. They are the ones who have really made the person I am today.

Implementing IS-95, the CDMA Standard, on TMS320C6201 DSP

by
Xiaozhen Zhang

Submitted to the
Department of Electrical Engineering and Computer Science

May 21, 1999

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Electrical Engineering
and Master of Engineering in Electrical Engineering

ABSTRACT

IS-95 is the present U.S. 2nd generation CDMA standard. Currently, the 2nd generation CDMA phones are produced by Qualcomm. Texas Instruments (TI) has ASIC design for Viterbi Decoder on C54x. Several of the components in the forward link process are also implemented in hardware. However, having to design a specific hardware for a particular application is expensive and time consuming. Thus, the possibility of the alternative implementations is of great interest to both customers and TI itself.

This research has achieved in successful implementation of IS-95 entirely in software on TI fixed-point DSP TMS320C6201, and met the real time constraint. IS-95 system, the industrial standard for CDMA, is a very complicated system and extremely computationally demanding. The transmission rate for an IS-95 system is 1.2288 Mcps. This research project includes all the major components of the demodulation process for the forward link system: PN Descrambling, Walsh Despreading, Phase Correction & Maximal Ratio Combining, Deinterleaver, Digital Automatic Gain Control, and Viterbi Decoder. The entire demodulation process is done completely in C. That makes it a very attractive alternative implementation in the future applications. It is well known that ASIC design is not only expensive and but also time consuming, programming in assembly is easier and cheaper, but programming in C is a much easier and efficient way out, in particular, for general computer engineers.

During the whole process, efforts have been devoted on developing various specific techniques to optimize the design for all the components involved. These developments are successfully achieved by making the best use of the following techniques: to simplify the algorithms first before programming, to look for regularity in the problem, to work toward the Compiler's full efficiency, and to use C intrinsics whenever possible. All these attributes together make the implementation scheme great for DSP applications. The benchmark results compare very well to the TI-internal hand scheduled assembly performance of the same type of decoders. The estimated percentage usage of all the components (excluding PN) is only 21.18% of the total CPU cycles available (4,000 K), which is very efficient and impressive.

Thesis Supervisor: Michael V. Bove
Title: Principal Research Scientist of MIT Media Lab

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1. INTRODUCTION

Wireless communication has been increasingly important as a new tool both for business and daily life. New and specific applications are growing at a much faster rate than ever before due to the increasing demand and sharp market competition. Thus, the traditional technique of developing products for specific applications through specific hardware designs or using assembly language is facing severe challenge because it is expensive, time consuming, and sometimes, at risk (at beginning). Finding better alternative implementations is of great interest not only for consumers, but also for manufacturers. As part of this goal, intensive research has been conducted to implement IS-95, the CDMA Standard, on TMS320C6201 DSP by C language.

1.1 Literature Review

1.1.1 The Wireless Communication Networks

For a long time, the wireless world has been confronted with the challenge of how to use its communication resources efficiently. The problem of providing the resources to multiple users while maintaining their mutual interference below an acceptable level has been central.

There are three major multiple access techniques employed in the existing wireless networks. Frequency Division Multiple Access (FDMA) and Time Division Multiple Access (TDMA) are the conventional techniques. Analog phones utilize the FDMA technology. Global System for Mobile (GSM) and IS-136 are standards for TDMA systems. IS-95 is the current U.S. 2^{nd} generation standard for Code Division Multiple Access (CDMA) and is a more recent development by Qualcomm in 1993 for

digital cellular applications. The 3rd generation standard is currently being proposed and is coming up soon.

In a FDMA system, users are assigned specific frequency bands that are disjoint from those of any other user. Each user has the sole right of using his or her frequency band for the entire call duration. Each user's signal is isolated by using pulse shaping filters that reduce out-of-band interference below an acceptable level. This effectively reduces the multiple access channel into many single point-to-point channels [Qualcomm, 1997]. The bandwidth and the Signal to Noise Ratio (SNR) of the channel help to determine its capacity. Larger bandwidth and higher SNR leads to higher capacity.

As an improvement to a FDMA system, a TDMA system shares many similar features with a FDMA system. However, rather than letting a single user occupy an assigned frequency band for the entire call duration, this frequency band is shared among several users. The idea of user channelization in the same frequency band is achieved through separation in time. Each user is only allowed to transmit through the band at predetermined time slots [Qualcomm, 1996]. The capacity of each channel is then further limited by the time allocated to each user.

The CDMA technique takes on a completely different approach. It does not attempt to allocate disjoint frequency or time resources to each user, but instead allocates all resources to all simultaneous users. CDMA users are channelized by uniquely assigned codes. The signals are separated at the receiver by using a correlator that uses the same code as the one for the desired user. After correlation (despreading), undesired signals contribute only as background noise and are usually modeled as additive white

Gaussian noise (AWGN) [Qualcomm, 1996]. A CDMA system has many advantages over a FDMA and a TDMA system. Its most significant contribution is the much more efficient use of the system's bandwidth, which will be discussed next.

1.1.2 Capacity Comparison

Most of the existing FDMA systems are analog systems, whereas the TDMA and CDMA systems are all digital. When it comes to voice transmission, digital systems have a natural edge over analog systems. A FDMA system needs 30KHz per channel for voice transmission, whereas due to data compression, only 10KHz per channel is needed for a TDMA system. It is easy to see that a TDMA system has a three times capacity gain when compared to that of a FDMA system. How does the capacity of a CDMA system compared to a TDMA system? CDMA has its basis in the spread spectrum technology. CDMA systems operate at a very low SNR, but use a very large bandwidth in order to provide acceptable capacity. CDMA's theoretical roots lie in the principles of Shannon's information theory. The capacity of a channel of band W perturbed by white thermal noise of power N when the average transmitter power is limited to P is given by

$$C = W \log (P+N)/N \qquad \text{(Eq. 1.1)}$$

This means that by sufficiently involved encoding systems we can transmit binary digits at the rate $W \log_2(P+N)/N$ bits per second, with arbitrarily small frequency of errors [Shannon, 1949]. Shannon's Capacity Equation relates capacity to both bandwidth and SNR. It shows that acceptable capacity can be achieved even at very low SNR, if adequate bandwidth is allocated. A cellular IS-95 channel (forward and reverse link) is a pair of frequencies with 1.25 MHz bandwidth 45 MHz apart.

The capacity of FDMA/TDMA cellular system is severely limited by its frequency reuse pattern. When multiple access in the same cell is achieved by using disjoint frequency bands, users in adjacent cells must also be provided disjoint frequency slots; otherwise interference between cells would become intolerable.

In a cellular system, this usually results in a frequency reuse pattern of 7 as shown in Figure 1.1 to provide a long enough distance between cells using the same frequency band so that the interference is diminished adequately due to path loss. In sectored cells that use three antennas to further divide up the cell, a reuse pattern of 21 is common. Basically, what this means is that at any time, only 1/7 of a carrier's frequency allocation could be used in any cell, and only 1/21 of it could be used in any cell sector. In a two-person conversation, when frequency and time resources are assigned exclusively to the users, these resources are further underutilized because each speaker is active less than half of the time.

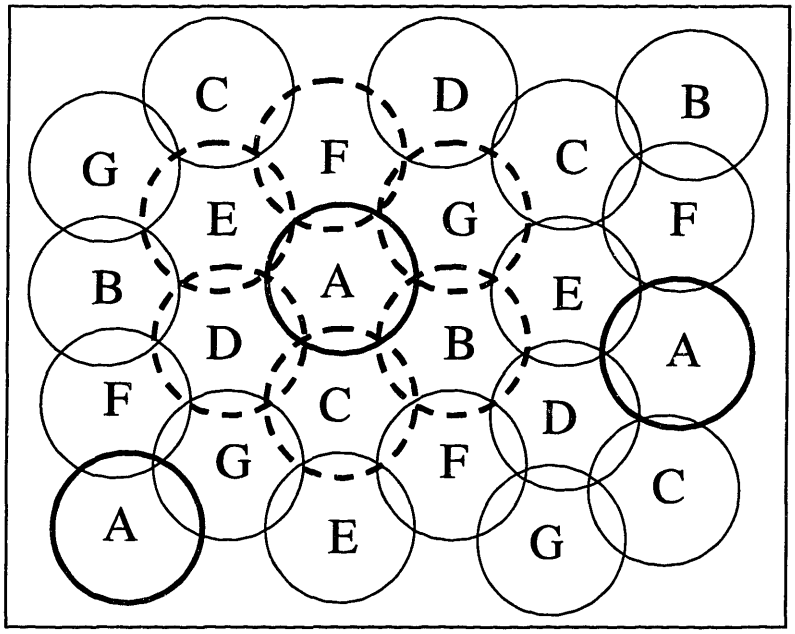


Figure 1.1. Cellular systems with a frequency reuse pattern of 7

At this point, it is easy to see the advantage of a CDMA system. The CDMA system can allocate all of its spectrum and time to all of its users in all cells simultaneously; and it can efficiently transform the pauses during a conversation into a decrease of the background noise. As shown in Figure 1.2, the same frequency spectrum can be used in all CDMA cells. So the overall capacity gain for a CDMA system is much higher. CDMA offers 5 to 7 times more capacity than a TDMA system; it offers 15 to 20 times more capacity when compared to a FDMA system [Qualcomm, 1996].

CDMA's multiple access capabilities and high bandwidth efficiency has established it as the leading technology in a bandwidth starved wireless communication world. The direction of the market is clear; the importance of CDMA technology is clear. The emerging 3rd generation wireless system being proposed is also based on the CDMA technology gives another demonstration of the market's emphasis on the CDMA technology.

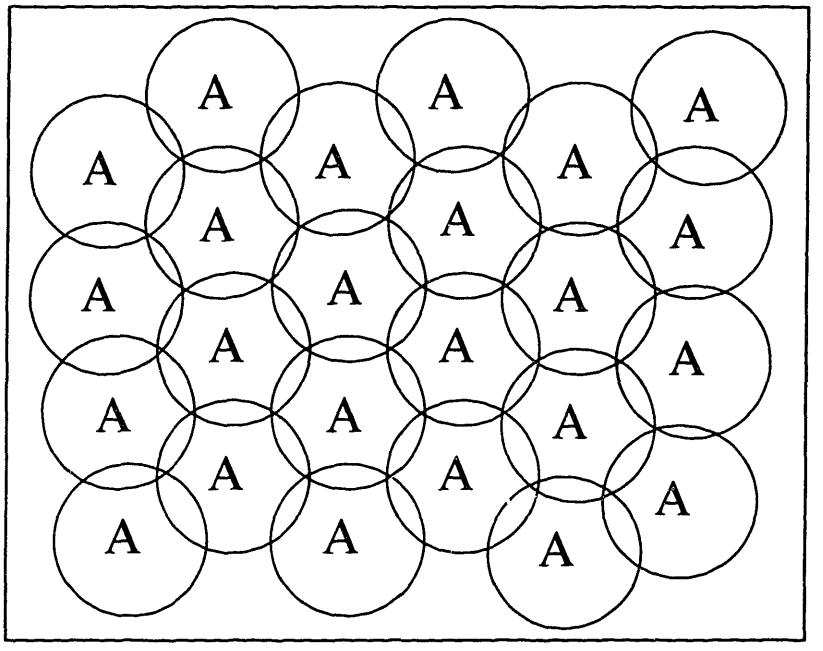


Figure 1.2. CDMA System frequency reuse

1.1.3 IS-95 Standard

In July 1993, the Telecommunications Industry Association (TIA) published IS-95 as the CDMA standard. The IS-95A revision was published in May [Qualcomm, 1996]. Subsequent revisions also include IS-95B and IS-95C. The IS-95 is the current U.S. 2nd generation standard, and the 3rd generation is coming up.

IS-95A specifies technical requirements that define a compatibility standard for wideband spread spectrum cellular mobile telecommunications. They ensure that a mobile station can obtain service in any cellular system manufactured according to this standard. IS-95A specifies requirements for both the mobile and base station, including message encryption and voice privacy, call flow, system layering, constants, retrievable and settable parameters, and the mobile station [Qualcomm, 1997].

Since the forward channel (base station to mobile communication) will be of primary interest to the research being proposed, some highlights will be presented here. Several types of digital signal processing are done to a signal prior to its transmission at the base station transmitter. First, the signal goes through a variable rate vocoder which produces a frame every 20 msec using Code Excited Linear Prediction (CELP) technique. There are two rate sets of vocoders. Cellular band can use both sets. Rate set 1 vocoder produces 192 bits per frame; rate set 2 produces 288 bits per frame. The quality of rate set 2 vocoder is superior to that of the rate set 1. For both rate sets, the variable rate vocoders can produce frames either at full, half, quarter or eighth rate. The full rate is 9.6 Kbps for rate set 1 and 14.4 Kbps for rate set 2. The frame rate depends on the voice

activity. Lower rates are generated by the vocoder for lower voice activity [Qualcomm, 1997].

The forward traffic channel supports both vocoder sets. Rate set 1 data are convolutionally encoded with a rate ½ encoder. Rate set 2 has a ½ rate encoder followed by puncturing to produce an effective coding rate of ¾. In addition to convolutional coding, the symbols are repeated when lower rate frames are produced by the vocoder so to maintain a constant symbol rate of 384 symbols per frame or 19,200 symbols per second regardless of the rate of the vocoder. Full rate frame does not have any repetition; half rate frame is repeated once; quarter rate frame is repeated three times; and eighth rate frame is repeated seven times. Symbol repetition reduces the "energy per symbol" requirement and leads to lower power transmission and lower interference to other users. The following block diagram (Figure 1.3) is an illustration of the forward traffic channel.

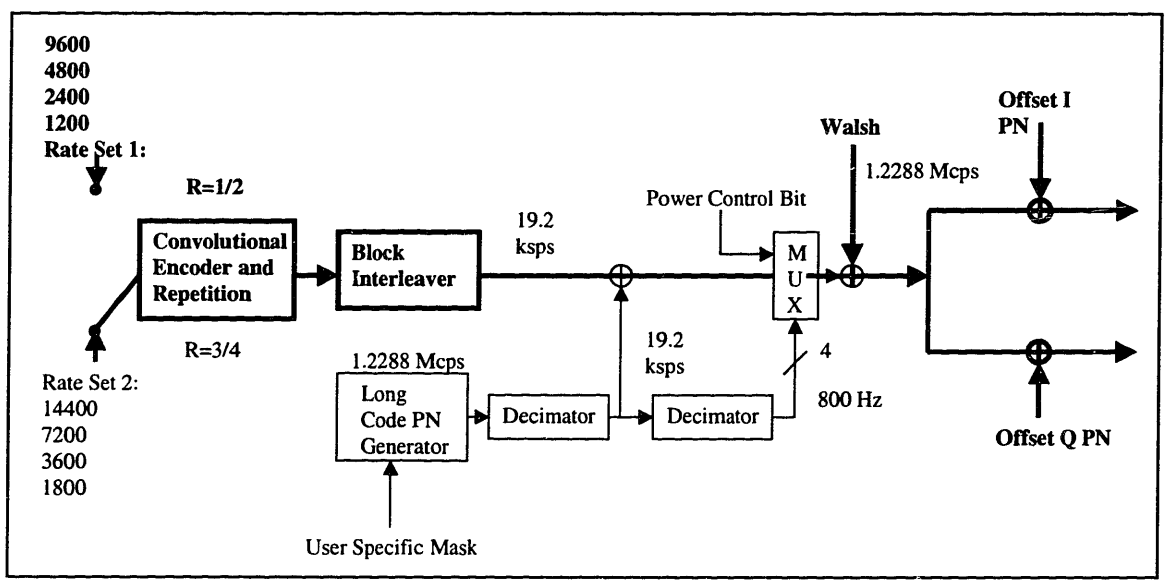


Figure 1.3. Block diagram of the generation of the forward traffic channel
(IS-95A: The CDMA Standard, p.3-8)

After the convolutional encoder, a block interleaver is then used to interleave the symbols. Interleaving is a jumbling of the symbols. Interleaving the symbols prior to transmission has the effect of "whitening" the channel: errors that occurred in bursts due to fading appear to be randomly scattered when the symbols are de-interleaved. This results in a more effective performance for the decoder since convolutional codes are useful when the errors are random and not in bursts. Interleaving is done at a block of the 20msec frame. There is no interleaving across the frame boundaries.

A CDMA system employs three pseudorandom noise (PN) sequences. The system has two short codes and one long code that are time-synchronized to midnight, January 6, 1980 (GPS time). All base stations and all mobiles use the same three PN sequences. The Long PN Code is used for spreading and scrambling. It repeats every 41 days (at a clock rate of 1.2288 Mcps). This provides a CDMA system with an inherent feature of voice privacy that greatly surpasses that provided by any FDMA or TDMA system. The two Short PN Codes are used for quadrature spreading; its unique offsets serve as identifiers for a cell or a sector and they are repeated every 26.67 msec (at a clock rate of 1.2288 Mcps) [Qualcomm, 1996]. An important property of a PN sequence is that time-shifted versions of the same PN sequence have very little correlation with each other.

After the symbol frame is interleaved, the Forward Traffic Channel is scrambled by the Long PN sequence. The 19,200 symbols per second are multiplied by the Long PN sequence that is also generated at 19,200 symbols per second.

The signal is then orthogonally spread using the Walsh codes. Within a sector, each traffic channel in the forward direction uses a unique Walsh code. This provides

isolation between channels within a sector due to the orthogonality condition of the codes. Each symbol is spread by all 64 chips of the Walsh code sequence. The Walsh codes are reused in every sector [Qualcomm, 1997].

After spreading by the Walsh code sequence, the forward traffic channel is scrambled over both quadratures. All of the information is sent into both quadratures (BPSK modulation). Each quadrature is spread using a short PN sequence with different time shifts for different sectors and cells. As mentioned previously, the two short PN sequences are used to isolate one sector from another. This enables the re-use of the Walsh codes in every sector [Qualcomm, 1997]. These two quadratures are then mapped into phase shifts of the carrier signal and sent to the transmitter (QPSK spreading).

1.1.4 Demodulating the Forward Traffic Channel

The IS-95 standard, however, contains no details on the receiver whose design is left to the manufacturer. The following is just a sketch of the demodulation procedure. After the signal is down converted from the carrier band to the baseband (the carrier band is 900-1000MHz for cellular and 1.9-2GHz for Personal Communication System (PCS)), filtering and A/D conversion are performed; and the signal is digitized. The mobile station implements a rake receiver design, which typically includes three to four finger correlators and a searcher correlator. The searcher identifies strong multipath arrivals and a finger is assigned to demodulate at the offset identified. The result is then coherently combined and passes through the PN de-scrambler and then the Walsh despreader. After Maximal Ratio Combining (MRC) for the signal paths of the different fingers, the signal is further processed by the de-interleaver and sent to the Viterbi

Decoder [Qualcomm, 1997]. Figure 1.4 is a block diagram illustration of the demodulating procedure.

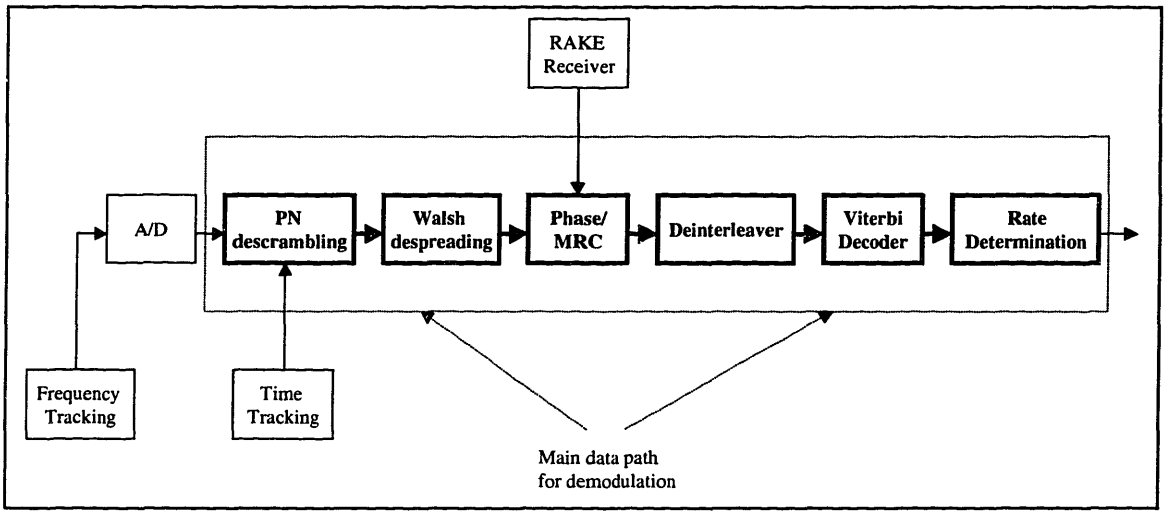


Figure 1.4. Demodulation of the Forward Traffic Channel

1.1.5 Previous Work

Qualcomm is the developer of the IS-95 standard. It is also currently the leader of the CDMA digital phone industry. Qualcomm is currently in production for both CDMA digital cellular and PCS phones. It is also planning on bringing to the market in the first half of 1999 pdQ smart phone that combines the state of art CDMA technology and Palm Computing® platform.

Previous work has also been done at Texas Instruments (TI) on its TMS320C54x DSPs to implement the demodulation process of the forward channels (base station to mobile communication). Because of power considerations due to the high data rate (1.2288 Mcps) and the lack of memory on the TMS320C54x DSP, PN and WALSH despreadings have been done in hardware on Application Specific Integrated Circuit

(ASIC). The TMS320C54x incorporates a special hardware unit to accelerate Viterbi metric-update computation. This compare-select-store unit with dual accumulators and a splittable ALU performs a Viterbi butterfly in four cycles. [Hendrix, 1996]

1.1.6 The TMS320C6201 Digital Signal Processor

TMS320C6201 is the most powerful fixed-point DSP currently available on the market. The 'C62x devices operate at 200 MHz (5-ns cycle time). It executes up to eight 32-bit instructions every cycle.

The 'C62x use the VelociTI architecture, a high-performance, advanced VLIW (very long instruction word) architecture, making these DSPs excellent choices for multi-channel and multifunction applications. [Texas Instruments, 1998]

The 'C62x have a 32-bit, byte-addressable address space (4 gigabytes). 'C6201 has 128Kbytes of on chip RAM. On chip memory is organized in separate data and program spaces. The family has two 32-bit internal ports to access internal data memory and a single internal port to access internal program memory. All internal memory is zero wait-state.

The 'C6x family has the industry's most efficient C compiler; its efficiency is three times the efficiency of other fixed-point DSP compilers, making the development of new products much easier and faster.

High performance, ease of use, and affordable pricing make the TMS320C6x family a great choice for the task undertaken.

1.2 Research Objectives

The objectives of this research are summarized as follows:

1) Implementing PN and WALSH Despreading on 'C6201;
2) Implementing Phase Correction/MRC on 'C6201;
3) Implementing the Deinterleaver function on 'C6201;
4) Implementing the Digital Automatic Gain Control function on 'C6201;
5) Implementing Viterbi Decoder on 'C6201; and
6) Benchmarking the system for meeting the real time constraint.

The efforts of this research have been to develop the demodulation procedure for all the major functions on the main data path of the forward channel. Specifically, this has involved the PN and Walsh despreading, Phase Correction & Maximal Ratio Combining (MRC), deinterleaving, Digital Automatic Gain Control (DAGC) and Viterbi decoding algorithm.

PN and Walsh despreading will be based on the correlation concept. When the original signals are a binary sequence, this corresponds to exclusive ORing (modulo two adding) the signals over time; when the signals are antipodal, i.e. in sequence of 1's and –1's, this correlation process corresponds to multiplying the signals over time.

In an additive white Gaussian noise (AWGN) channel, the received signal is detected by using a matched filter. The detection process is essentially the projection of one vector onto another. To maximize the value of the result (maximize the difference between the possible hypotheses and minimize the error probability), the phase between the two vectors should be zero. The phase correction & MRC algorithm corrects the

phase uncertainty of the received signal while at the same time it optimally combines the different signal multipaths to provide time diversity against fading and to optimize the effective SNR [Papasakellariou, 1998].

As discussed earlier, interleaving the symbols prior to transmission has the effect of "whitening" the channel. The deinterleaving process simply rearranges the jumbled symbols into their correct order as to prepare them for the decoder.

A Digital Automatic Gain Control (DAGC) is needed to maintain the input signal to Viterbi Decoder within a certain dynamic range. The data processing of PN descrambling, Walsh despreading, and MRC leaves the output signal of the Deinterleaver to be represented using 28 or 32 bits. This is a much larger dynamic range than what is traditionally supported by the Viterbi Decoder. DAGC weighs the input signal over one frame of data and limits the dynamic range that is represented using only 5 bits.

Convolutionally encoded data is decoded through knowledge of the possible state transitions, created from the dependence of the current symbol on past data. The allowable state transitions are concisely represented by a trellis diagram. Convolutional codes are decoded by using the trellis to find the most likely sequence of codes. The Viterbi Decoding Algorithm simplifies the decoding task by limiting the number of sequences examined. The most likely path to each state is retained for each new symbol. The Viterbi Decoding Algorithm includes two functions: metric update and traceback. Because each state has two or more possible input paths, the accumulated distance is calculated for each input path. The path with the minimum accumulated distance is selected as the survivor path. An indication of the path and the previous Delay State is stored to enable reconstruction of the state sequence from a later point. The actual

decoding of symbols into the original data is accomplished by tracing the maximum likelihood path backward through the trellis. The original data is reconstructed from the states sequence [Hendrix, 1996].

It is expensive and risky (at first) having to design a specific hardware set to support a specific application. With the appearance of more and more powerful processors, it is possible that we could rely less and less on having special hardware to get special task done. The demodulation process for the forward traffic channels is very computationally intensive. For example, the IS-95A implementation of the Viterbi Decoding algorithm, which has a constraint length 9 convolutional encoder, requires 128 butterfly calculations for each metric update. For one frame of symbols, metric update needs to be done 192 times. Since the IS95-A standard allows for various rate vocoder, typically, four Viterbi decoders need to be implemented for each frame of data for each rate set. The amount of calculations is obviously nontrivial.

DSPs are optimized for additions and multiplications. With an especially powerful DSP, it might be possible to implement a very complicated system, such as the demodulation of the forward channels, entirely in software. If this were the case, we could dramatically cut down the development cost for a new system and bring the products to market much faster, although the DSP solution will require more power than a full ASIC one.

Thus, this research has been concentrated on implementing all the major functions for demodulating the forward traffic channel's main data path on a single TI TMS320C6201 DSP to see if it could be capable of meeting the real time constraint of

the system. "C" is the only language for implementing these functions. The result of this work is of practical importance to TI and many people working in this field.

CHAPTER 2. WALSH DESPREADING, PHASE CORRECTION & MAXIMUM RATIO COMBINING

2.1 Walsh Despreading

2.1.1 Considerations for Software Design

Walsh codes are orthogonal codes of length 2^n and there are 2^n such codes. In IS-95, there are 64 of them. They are used in the IS-95 forward link system to separate different users from the same cell or sector. The IS-95 standard specifies that the sync channel contains the information about the specific Walsh code used for spreading. The Rake Receiver then uses this information for despreading.

The Walsh Despreading function unit must be able to support the despreading of any of the 64 users. It is not known in advance which Walsh code would be the one needed for despreading. Therefore, the system needs to have the knowledge of all 64 Walsh sequences. This leaves the design choice of either having all 64 sequences stored in on-chip memory; or to have the function unit be able to generate the requested Walsh code efficiently. Each Walsh code contains 64 chips. Storing all 64 sequences would be a huge consumption of the valuable on-chip memory space. Although theoretically, it could be done; practically, it raises a lot of questions including efficiency and on-chip memory space. Therefore, going for the alternative route of finding the efficient way to generate the desired Walsh sequence immediately becomes the task of great importance. It is obvious that simply following the standard procedures of programming and without solving this challenge, the generated program would not only waste a huge amount of on-chip memory space, but also eventually affect the performance.

2.1.2 Optimization of Design Strategy

The C code developed for the Walsh Despreading algorithm assumes that the Rake Receiver has already determined the Walsh sequence need to be generated. The function unit simply takes the code sequence number and generates the appropriate Walsh code, then uses the Walsh code to demodulate the input sequences.

For the forward link generation, input symbols of data rate 19.2 Kbps are spreaded by the Walsh code to a data rate of 1.2288 Mcps. In other words, one input symbol is spreaded by all 64 chips of a Walsh code. The Walsh code sequence used is unique to the user of the cell or the sector. The same Walsh code is used to spread all the symbols of the same source. At the receiver's end, the same Walsh code sequence is used for despreading. The function implemented assumes that the Rake Receiver has already resolved the timing issue, so that the input chips are synchronized with the Walsh chip set.

The Walsh Despreding unit consists of two parts: the Walsh code sequence generating function generator() and the despreading portion. Function generator() takes the desired Walsh code number as its parameter and generates the appropriate Walsh sequence. This promoted the initiative to make the best use of the unique characteristic of the Walsh code, its regularity.

The Walsh code table is a big 64×64 matrix as shown in Table 2.1. The appearance gives the impression that it contains too many parameters. As discussed above, following the standard procedures by taking the whole 64x64 matrix would result in occupying huge amount of valuable on-chip memory space. Is there any way to simplify the algorithm and optimize the implementing design? A careful study of the

Table 2.1. IS-95 Walsh Code Functions

	0 1	2 3	4 5	6 7	8 9	10 11	12 13	14 15	16 17	18 19	20 21	22 23	24 25	26 27	28 29	30 31	32 33	34 35	36 37	38 39	40 41	42 43	44 45	46 47	48 49	50 51	52 53	54 55	56 57	58 59	60 61	62 63
0	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1
1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1
2	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1
3	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	-1 1
4	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1
5	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	-1 1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1
6	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1
7	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1
8	1 1	1 1	1 1	1 1	-1 -1	-1 -1	-1 -1	-1 -1	1 1	1 1	1 1	1 1	-1 -1	-1 -1	-1 -1	-1 -1	1 1	1 1	1 1	1 1	-1 -1	-1 -1	-1 -1	-1 -1	1 1	1 1	1 1	1 1	1 1	1 1	-1 -1	-1 -1
9	1 -1	1 -1	1 -1	1 -1	-1 1	-1 1	-1 1	-1 1	1 -1	1 -1	1 -1	1 -1	-1 1	-1 1	-1 1	-1 1	1 -1	1 -1	1 -1	1 -1	-1 1	-1 1	-1 1	-1 1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	-1 1	-1 1
10	1 1	-1 -1	1 1	-1 -1	-1 -1	1 1	-1 -1	1 1	1 1	-1 -1	1 1	-1 -1	-1 -1	1 1	-1 -1	1 1	1 1	-1 -1	1 1	-1 -1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1
11	1 -1	-1 1	1 -1	-1 1	-1 1	1 -1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	1 -1	-1 1	1 -1	-1 1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1
12	1 1	1 1	-1 -1	-1 -1	-1 -1	-1 -1	1 1	1 1	1 1	1 1	-1 -1	-1 -1	-1 -1	-1 -1	1 1	1 1	1 1	1 1	-1 -1	-1 -1	-1 -1	-1 -1	1 1	1 1	1 1	1 1	-1 -1	-1 -1	-1 -1	-1 -1	1 1	1 1
13	1 -1	1 -1	-1 1	-1 1	-1 1	1 -1	1 -1	1 -1	1 -1	1 -1	-1 1	-1 1	-1 1	-1 1	1 -1	1 -1	1 -1	1 -1	-1 1	-1 1	-1 1	-1 1	1 -1	1 -1	1 -1	1 -1	-1 1	-1 1	-1 1	-1 1	1 -1	1 -1
14	1 1	-1 -1	-1 -1	1 1	-1 -1	1 1	1 1	-1 -1	1 1	-1 -1	-1 -1	1 1	-1 -1	1 1	1 1	-1 -1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1
15	1 -1	-1 1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	-1 1	1 -1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1
16	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1
17	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1
18	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1
19	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1
20	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	1 1	1 1	1 1	1 1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	1 1	1 1	1 1	1 1	1 1	1 1
21	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1
22	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1
23	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1
24	1 1	1 1	1 1	1 1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1
25	1 -1	1 -1	1 -1	1 -1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1
26	1 1	-1 -1	1 1	-1 -1	-1 -1	1 1	-1 -1	1 1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	-1 -1	1 1	-1 -1	1 1	-1 -1
27	1 -1	-1 1	1 -1	-1 1	-1 1	1 -1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1
28	1 1	1 1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1
29	1 -1	1 -1	-1 1	-1 1	-1 1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1
30	1 1	-1 -1	-1 -1	1 1	-1 -1	1 1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	1 1	-1 -1	-1 -1	1 1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1
31	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	1 -1	-1 1	-1 1	1 -1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1
32	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1
33	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1
34	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1
35	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1
36	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1
37	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	-1 1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1
38	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1
39	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1
40	1 1	1 1	1 1	1 1	-1 -1	-1 -1	-1 -1	-1 -1	1 1	1 1	1 1	1 1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	1 1	1 1	1 1	1 1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1
41	1 -1	1 -1	1 -1	1 -1	-1 1	-1 1	-1 1	-1 1	1 -1	1 -1	1 -1	1 -1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	1 -1	1 -1	1 -1	1 -1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1
42	1 1	-1 -1	1 1	-1 -1	-1 -1	1 1	-1 -1	1 1	1 1	-1 -1	1 1	-1 -1	-1 -1	1 1	-1 -1	1 1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1
43	1 -1	-1 1	1 -1	-1 1	-1 1	1 -1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1
44	1 1	1 1	-1 -1	-1 -1	-1 -1	-1 -1	1 1	1 1	1 1	1 1	-1 -1	-1 -1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	1 1	1 1	-1 -1	-1 -1	-1 -1	-1 -1	1 1	1 1	1 1	1 1	-1 -1	-1 -1
45	1 -1	1 -1	-1 1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1
46	1 1	-1 -1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1
47	1 -1	-1 1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1
48	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1
49	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1
50	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1
51	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1
52	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1
53	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1
54	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1
55	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1
56	1 1	1 1	1 1	1 1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	-1 -1	1 1	1 1	1 1	1 1	1 1	1 1	-1 -1	-1 -1	-1 -1	-1 -1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	-1 -1	-1 -1	-1 -1
57	1 -1	1 -1	1 -1	1 -1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	-1 1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	-1 1	-1 1	-1 1	-1 1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	1 -1	-1 1	-1 1	-1 1
58	1 1	-1 -1	1 1	-1 -1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1	1 1	-1 -1
59	1 -1	-1 1	1 -1	-1 1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1
60	1 1	1 1	-1 -1	-1 -1	-1 -1	-1 -1	1 1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1
61	1 -1	1 -1	-1 1	-1 1	-1 1	1 -1	1 -1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1
62	1 1	-1 -1	-1 -1	1 1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1	1 1	-1 -1	-1 -1	1 1
63	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	-1 1	1 -1	1 -1	-1 1	-1 1	1 -1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1	1 -1	-1 1

Walsh code reveals that there is a great deal of regularity within the Walsh code as discussed below:

- Table 2.1 could also be viewed as consisting of four smaller 32×32 blocks. Observing carefully demonstrates that the upper right corner block, the lower left corner block, and the upper left corner all consist of the same 32×32 block. The lower right corner is the exception and consists of the negative of the other blocks.
- Similarly, the 32×32 block can also be viewed as consisting of four smaller 16×16 blocks. These four 16×16 blocks exhibit the same pattern as their larger counterparts.
- This regularity can be observed all the way to the elementary 2×2 blocks where the rudimentary component blocks are single 1 and –1's. In concise vector terms, a Walsh code set of length 2N can be constructed as w_2N = [w_N w_N; w_N –w_N].

Now, if all these regularities could be made the best use, the program would no longer need to treat a 64x64 matrix for the Walsh Spreading. Instead, a much smaller matrix would be handled, which could greatly save the on-chip memory space, and eventually improve the overall performance of the implemented program.

By studying the Walsh code in more details, it reveals that these regularities of Walsh codes can be put into more concrete terms and used toward programming optimization. The followings have been observed and considered:

- Following the convention and naming the Walsh Functions from #0 to #63, for all even Walsh Function numbers, the first two elements of the array are

all equal to 1; for all odd Walsh Function numbers, the 1st element is 1 and the 2nd element is equal to -1.

- Considering the 4×4 blocks, the lower right corner is the negative of the others. Therefore, for Walsh Function number%4 <= 1, the 3rd and 4th elements of the array are the exact replica of the first two; on the other hand, for Walsh function number%4 is equal to 2 or 3, the 3rd and 4th elements negate the first two elements.
- Following the path and considering the 8×8 blocks, the lower right corner is the negative of the other corner components. Therefore, for Walsh Function number%8 <= 3, the 5th through the 8th elements of the array are the exact replica of the first four elements. For Walsh Function number%8 that is equal to 4, 5, 6, or 7, the 5th through the 8th elements of the array are the negative of the first four elements.
- By considering larger and larger blocks, the 9th to 16th elements, 17th to 32nd elements, 33rd to 64th elements of the Walsh Function sequence can be generated accordingly.

Taking all those regularities into considerations, it can be seen that a 64x64 Walsh code matrix now would be treated in a much smaller scale, which is certain to benefit the whole programming process.

2.1.3 Despreading

The despreading part of the code aims to absorb the effect of Walsh spreading and cutting down the data rate back to 19.2 Ksps (The input to the Walsh Despreading function has a data rate of 1.2288 Mcps). Assuming the timing is synchronized, the

despreading function simply correlates the input chips with the appropriate Walsh sequence and sums the result over a 64-chip range. The dynamic range of the outputs of the Walsh despreading unit (at the receiver's end) is 6 bits larger than its inputs due to the summation. The fast increase of the data's dynamic range brings up the need for later processing to limit the gain.

2.1.4 Summary

The Walsh Despreading has been successfully implemented in C on C6201. The experience from this implementation demonstrates that it is critical for programming to simplify the algorithm first, and look for regularity in the problem of concern. By taking the advantage of the regularity of the Walsh code, no memory storage is required for storing the code sequences in advance. This is a substantial saving for the valuable on-chip memory compared to taking a 64x64 Walsh code matrix. Besides, appropriate code sequence is generated real time based on the code sequence number requested by the user, which assumingly is already determined by the Rake Receiver.

Due to its high input chip rate (1.2288 Mcps), the Walsh Despeading functions takes up a relatively large amount of processing time. The benchmark result for processing one frame of input signal (24,576 chips) is about 63,600 CPU cycles; this is approximately 1.59% of the CPU time.

2.2 Phase Correction and Maximal Ratio Combining (MRC)

2.2.1 Design Strategy

In an additive white Gaussian noise (AWGN) channel, the received signal is detected by using a matched filter. The detection process is essentially the projection of one vector onto another. To maximize the value of the result (maximize the difference between the possible hypotheses and minimize the error probability), the phase between the two vectors should be zero. The phase correction & MRC algorithm corrects the phase uncertainty of the received signal while at the same time it optimally combines the different signal multipaths to provide time diversity against fading and to optimize the effective SNR [Papasakellariou, 1998].

In an IS-95 system, the transmission of a strong, unmodulated pilot signal offers the possible solution for both phase correction and optimum path combining. The pilot signal is transmitted synchronously with the information signal. The two signals have the same phase when they arrive through the same channel path. Multiplying the information signal with the complex conjugate of the pilot signal can get rid of the phase uncertainty, i.e., $(Ae^{j\theta})\times(Be^{-j\theta}) = A*B$, which has zero phase regardless of what the actual value of θ is.

Both the received pilot signal and the information signal behave like pseudonoise due to the I and Q PN scrambling code. As a result, signal paths that are separated by more than one chip interval appear uncorrelated. Combining such paths provides time diversity against fading and increases the effective SNR. With MRC, each path is scaled according to its SNR prior to combining. The necessary path weighting is accomplished through the same operation as for phase correction [Papasakellariou, 1998]. Multiplying

the information signal with its pilot conjugate results in a functional unit that both corrects the phase uncertainty and does MRC.

The pilot signal and the information signal are received over their quadratures. Both the I and Q components of the pilot signal are summed over a period of 16 symbols for phase correction and MRC with I being the real part and Q being the imaginary part of the signal. The pilot is summed in order to reduce the signal variations due to fading and noise. The duration over which the phase remains constant depends on the mobile speed. For the highest mobile speeds considered in IS-95, the phase remains practically the same over 20-24 symbols in a cellular system, and about 10-12 symbols in a PCS system. This phase stability property of the channel limits the number of symbols that could be used for averaging (or summation). Increasing the number of symbol periods much beyond the phase stability range of the channel improves the amplitude estimate for MRC but degrades the instantaneous phase estimate for phase [Papasakellariou, 1998]. This particular implementation has been chosen to sum over 16 symbols for PCS frequencies; similar performance could be achieved if another value close to 16 had been used. Also, the symbol summation range is allowed to go twice as large for cellular frequencies.

Summing the pilot signal for its smoothing effect naturally introduces a new problem for consideration. Averaging could be done both before and after the "on time" symbol period. For example, the pilot averaging over N symbol periods can begin from N-1 symbol periods before the "on time" traffic signal (causal) or it can begin before and continue after the "on time" period for a total of N symbol periods (noncausal). This Phase Correction & MRC algorithm is implemented as a non-causal system. A detailed

discussion of the performance comparison of a noncausal system over a causal system is presented in IS-95B Algorithm Description Document [Papasakellariou, 1998]. Non-causal implementation provides a delay to the throughput of the system but provides a 0.1 dB performance gain over a causal system.

This implementation creates two circular buffers of size 16 each for the I and Q components of the pilot signal. It also creates two circular buffers of size 7 that are used to store the delayed I and Q symbols of the information signal. Two variables avg_I and avg_Q are used to store the sum of the pilot's I and Q components over 16 symbol periods. The buffers and the sum variables are all initialized to zero. The oldest element of the pilot buffer is subtracted from the sum variable while new element is read and written to its place. The I component of the signal buffer is multiplied with avg_I whereas the Q component of the signal buffer is multiplied with avg_Q. The sum of the two forms the desired output. This is a simplification of the mathematics.

$$(Ae^{j\theta})\times(Be^{-j\theta}) = (A\cos\theta + jA\sin\theta)\times(B\cos\theta - jB\sin\theta)$$

$$= A\cos\theta \times B\cos\theta + A\sin\theta \times B\sin\theta \qquad \text{(Eq. 2.1)}$$

Here the cosine terms are the I terms of the pilot and information signal; and the sine terms are the Q terms. The element of the signal buffer that is multiplied by the sum variable is actually also the oldest element of the array. After the multiplication, new information symbol is read and written at its place. This implementation scheme creates a 7-symbol-delay to the throughput; it allows the 16-element sum to include 7 future values of the "on-time" pilot signal for averaging, providing the desired noncausal performance gain.

2.2.2 Summary

The Phase Correction & MRC algorithm involves the processing of the data frame for the I and Q branches of both pilot and information signals. There are 384 symbols per frame of data. The benchmark result for its implementation is about 25,400 CPU cycles over one frame of data. This is about 0.64% of the processor time.

CHAPTER 3. IMPLEMENTING DEINTERLEAVER AND DAGC

3.1 Implementing the Deinterleaver

3.1.1 The Deinterleaver

The Interleaver jumbles around the symbols transmitted so that transmission errors that occur in bursts are spread out after the symbols are put back to the original order. The interleaver has a "whitening" effect for the communication channel and is important for the forward traffic channel generation. The Deinterleaver's job is to rearrange the received data frame and to put the transmitted symbols back to their correct order. The primary task in implementing such an algorithm is to efficiently generate the sequence array that rearranges the input sequences.

3.1.2 Discover the Regularity for Optimization

As in the other implementations discussed above, it is critical to discover the regularity or to simplify the algorithm of the Deimterleaver in order to optimize the program. Intensive efforts have been initiated for this intention, and some very useful regularity has been discovered, and applied in the programming.

Table 3.1 shows the Interleaver sequence for a Full Rate data frame. The sequence is to be read vertically from left to right, i.e., the interleaver takes the 1^{st}, 65^{th}, 129^{th} ... output of the convolutional encoder as its 1^{st}, 2^{nd}, 3^{rd} ... output. The Deinterleaver's job obviously is to take its 1^{st}, 2^{nd}, 3^{rd} ... input symbol and rearranges them to be the 1^{st}, 65^{th}, 129^{th} output symbol.

One of the ways to achieve this is to generate the interleaver array which contains the sequence of Table 3.1 in its exact order (called it order[]), and let output[order[i]-1] = input[i] where i stands for the ith element of the input array. What is important here is to

Table 3.1. IS-95 Full Rate Frame Interleaver

1	9	5	13	3	11	7	15	2	10	6	14	4	12	8	16
65	73	69	77	67	75	71	79	66	74	70	78	68	76	72	80
129	137	133	141	131	139	135	143	130	138	134	142	132	140	136	144
193	201	197	205	195	203	199	207	194	202	198	206	196	204	200	208
257	265	261	269	259	267	263	271	258	266	262	270	260	268	264	272
321	329	325	333	323	331	327	335	322	330	326	334	324	332	328	336
33	41	37	45	35	43	39	47	34	42	38	46	36	44	40	48
97	105	101	109	99	107	103	111	98	106	102	110	100	108	104	112
161	169	165	173	163	171	167	175	162	170	166	174	164	172	168	176
225	233	229	237	227	235	231	239	226	234	230	238	228	236	232	240
289	297	293	301	291	299	295	303	290	298	294	302	292	300	296	304
353	361	357	365	355	363	359	367	354	362	358	366	356	364	360	368
17	25	21	29	19	27	23	31	18	26	22	30	20	28	24	32
81	89	85	93	83	91	87	95	82	90	86	94	84	92	88	96
145	153	149	157	147	155	151	159	146	154	150	158	148	156	152	160
209	217	213	221	211	219	215	223	210	218	214	222	212	220	216	224
273	281	277	285	275	283	279	287	274	282	278	286	276	284	280	288
337	345	341	349	339	347	343	351	338	346	342	350	340	348	344	352
49	57	53	61	51	59	55	63	50	58	54	62	52	60	56	64
113	121	117	125	115	123	119	127	114	122	118	126	116	124	120	128
177	185	181	189	179	187	183	191	178	186	182	190	180	188	184	192
241	249	245	253	243	251	247	255	242	250	246	254	244	252	248	256
305	313	309	317	307	315	311	319	306	314	310	318	308	316	312	320
369	377	373	381	371	379	375	383	370	378	374	382	372	380	376	384

Reading

Reading

be able to generate this Interleaver array efficiently. There are many ways to generate this array. However, by recognizing the inherent pattern of the Interleaver array, this array generator has been implemented with an especially simple form and it runs very efficiently on C6201.

3.1.3 The Rule of "64"

Looking at Table 3.1 carefully and innovatively, it is not difficult to discover the regularity in its data array. The 2nd element read is 64 greater than the 1st element, so is the 3rd to the 2nd one, the 4th one to the 5th one. This trend holds for the entire array except for every other 6th element. However, this simplifies the task significantly since it is only necessary to keep track of every 7th element of the array and all the other elements can be easily generated by adding a multiple of 64's to the head of the hexad. Thus, keeping track of only 64 symbols is enough to generate the entire array. To make it easy to remember, it is called the Rule of "64" in this project.

3.1.4 The Rule of "32-16-48"

The second regularity of the array is called the Rule of "32-16-48" for convenience in this project. Looking at all the columns of Table 3.1 can discover this rule. The 7th element is 32 greater than the 1st element of the column; the 13th element is 16 greater than the 1st element; the 19th one is 48 greater than the 1st element; and this is true for all the columns. This observation simplifies even further the number of array elements that need to be kept track of. Basically, if the 1st element of each column is known, then the entire column can be easily generated. This boils down to the need of keeping track of only the elements in the 1st row that contains only 16 symbols.

3.1.5 The Rule of "1-3-2-4"

The third unique characteristic of the array discovered during the programming is less obvious but is just as useful. Again, for convenience, this is named the Rule of "1-3-2-4" in this project. For a better understanding, this rule is discussed in detail as follows.

It is very helpful to notice that 1 is the 1^{st} element of the first row; 2 is the 9^{th} element of the same row; 3 is the 5^{th} element of the row; and 4 is the 13^{th} element of the row. Further more, each of these elements forms a quartet of its own with a repeating regularity. The row element that immediately follows 1 is 8 greater than it; the 3^{rd} element is 4 greater than the 1^{st} element; and the 4^{th} element is 12 greater than it. Viewing 1, 3, 2, 4 as the head of the quartet, this regularity appears in all four-quartet groups. And this first row can be generated by only two simple for-loops. Thus, the efficiency of the program is substantially increased, and the running time is greatly reduced.

3.1.6 Summary

It is very important to carefully analyze the data pattern in this Deinterleaver implementation. With the best use of the three discovered rules, i.e., the three unique characteristics, the Interleaver array has been generated very efficiently with an especially easy form. This particular implementation does not require any prior memory storage for the array and has a very small code size. Of course, the simpler the code, the faster it runs. All these attributes together make this implementation scheme great for DSP applications. The benchmark result for the Deinterleaver function to process one frame of data (384 symbols) is 1,440 CPU cycles; that is only about 0.036% of the

processor time, a very short time indeed. As noticed during the programming process, without using these rules, the efficiency and performance would be substantially lower than what has been achieved now.

3.2 Implementing DAGC on C6201

3.2.1 DAGC and Its Regular Implementation Technique

A Digital Automatic Gain Control (DAGC) is needed to maintain the input signal to Viterbi Decoder within a certain dynamic range. The data processing of PN descrambling, Walsh despreading, and MRC leaves the output signal of the Deinterleaver to be represented using 28 or 32 bits. This is a much larger dynamic range than what is traditionally supported by the Viterbi Decoder. DAGC weighs the input signal over one frame of data and limits the dynamic range that is represented using only 5 bits.

Figure 3.1. Schematic DAGC Algorithm Diagram before Simplicfiation

- For $X_1 \ldots X_{384}$, $|X_i|$ --> $\ln|X_i|$ --> $\ln|X_i| / \ln 2$ --> int $(\ln|X_i| / \ln 2)$

  --> $\sum_1^{384}$ int $(\ln|X_i| / \ln 2)$ --> int$(a \times \sum_1^{384} (..))$ -->

  $2^{\text{int}(a \sum_1^{384} (..))}$ --> f --> f$\times X_i$ --> S <--- Input to Viterbi.

- $a = -1/384$

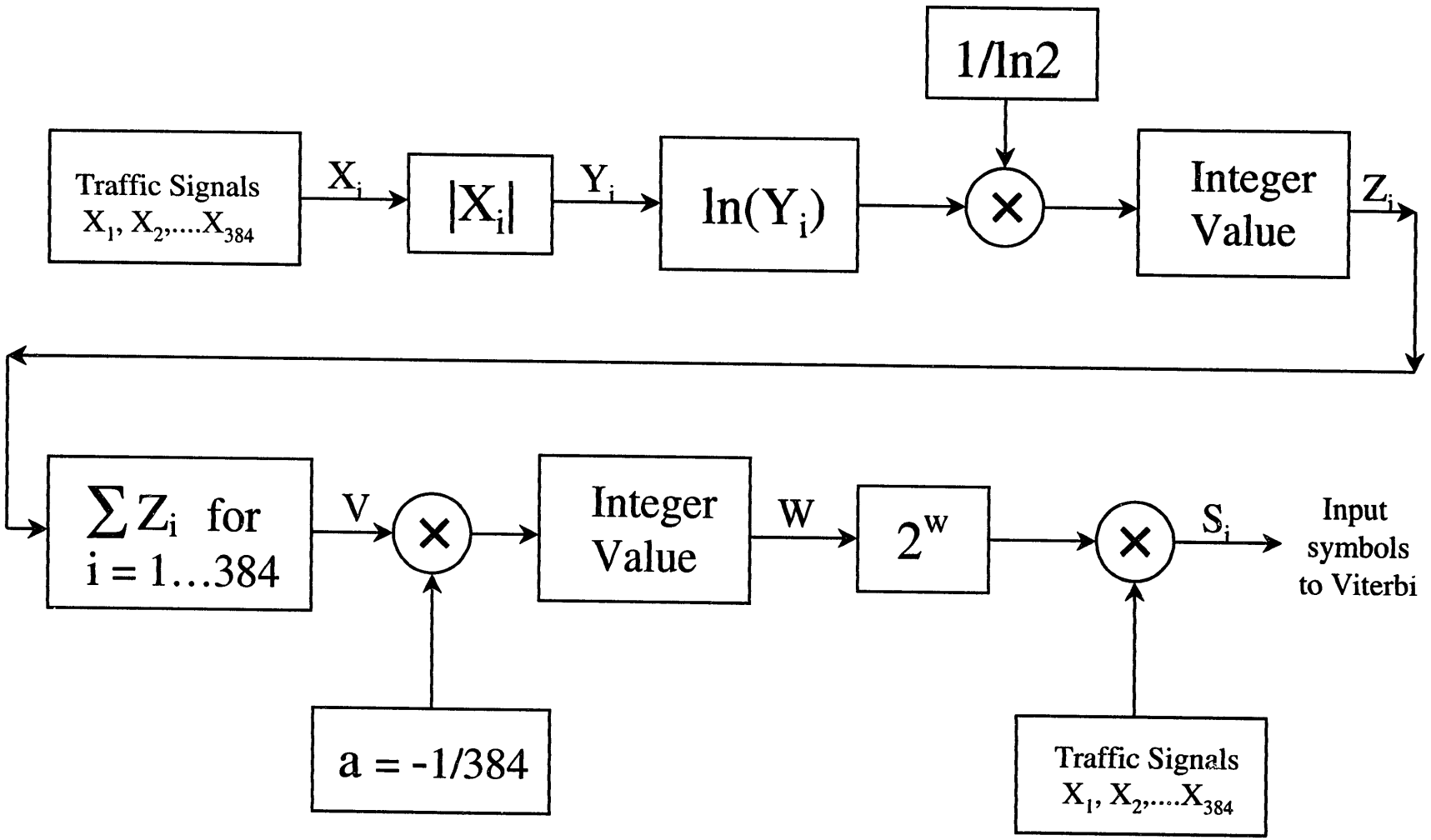
The schematic DAGC algorithm for implementation is presented in Figure 3.1.

The required signal processing of the DAGC unit is described in Figures 3.2 [Papasakellariou, 1998]. As illustrated in Figures 3.1 and 3.2, the signal processing is very complicated, and requires substantial amount of floating point calculations over a sequence of 384 input symbols. Although implementing floating point calculation is possible on a powerful fixed point DSP such as C6201, this procedure requires calling a floating point library that consumes a lot of CPU cycles. The immediate negative consequence is a significant reduction in the processing speed. In addition, large amounts of floating point calculations would eventually increase the error range of the results.

3.2.2 Search for New Approaches

Based on the above discussion, the primary task in implementing such a complicated algorithm of DAGC is to look for alternative approaches that could simplify the processing, and to minimize the floating point calculations, if possible. Without significant changes to the specific processing algorithm, it is obvious that the implemented program would be slow and error prone. In order to achieve that goal, intensive study has been conducted on simplifying the algorithm to minimize floating point calculations, and using C intrinsics whenever possible, which are very useful functions for the project. With the combination of all these efforts, the implemented DAGC has achieved satisfactory results. The DAGC algorithm, though apparently looks quite complicated at the very beginning, turns out to have a rather simple solution that requires very little floating point calculations. The alternative approaches are summarized in Figure 3.3 for comparison, and some details are discussed below.

Figure 3.2. Required Processing of DAGC



3.2.3 Use C Intrinsics to Improve Programming

The C intrinsic functions are very useful here. C intrinsic functions are compiler built-in assembly functions that can be called directly by a C procedure. The C6x compiler supports over thirty C intrinsic functions. Quite often, the C intrinsic functions can be used to process a task that would be very awkward to implement in pure C. In this case, the C intrinsic _abs() is used to take the saturated absolute value of X_i for all 384 input symbols.

3.2.4 Simplify the Algorithm Substantially

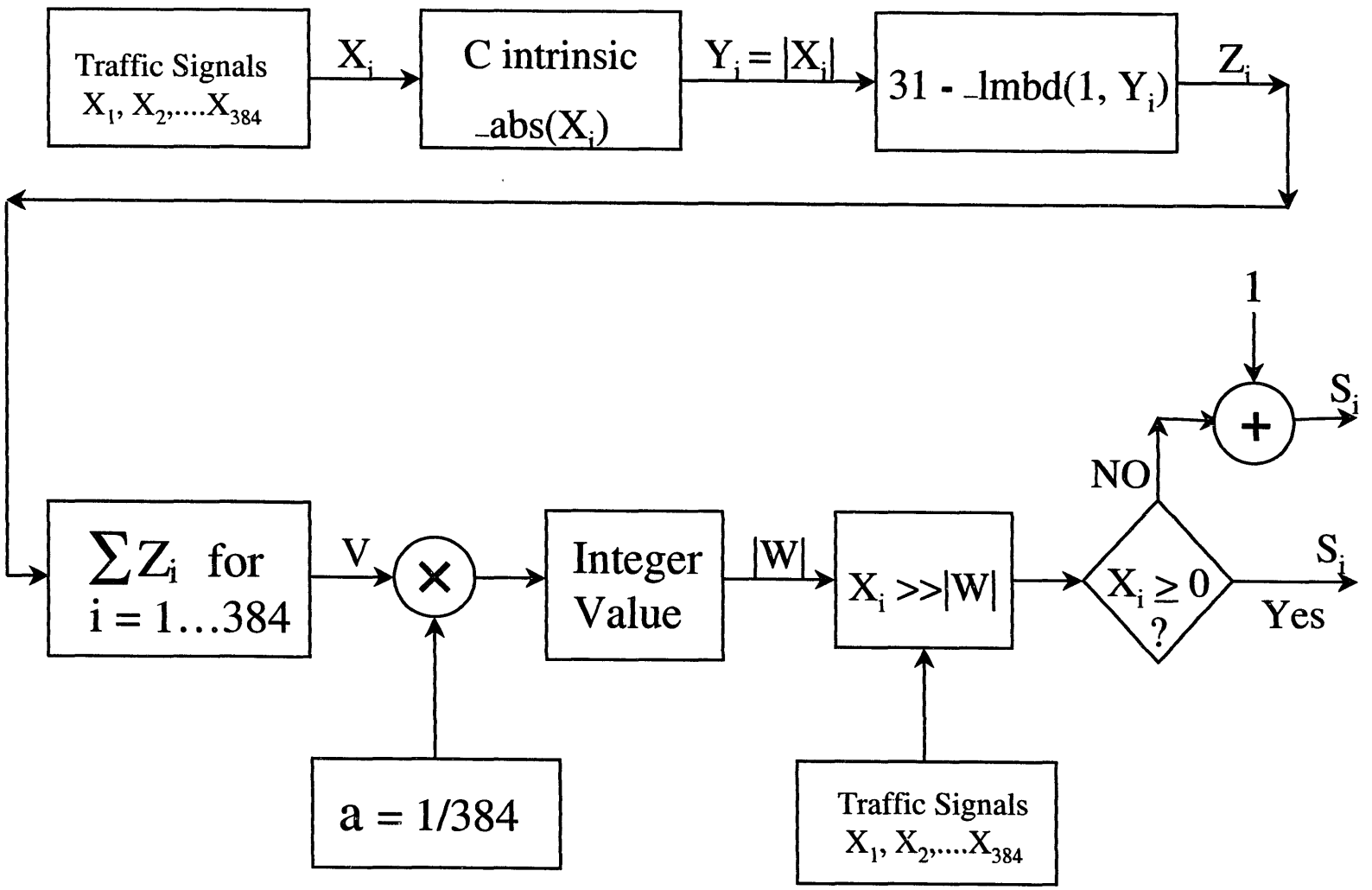
A lot of efforts have been taken to substantially simplify the algorithm of DAGC. The results are shown in Figures 3.3 and 3.4. The schematic DAGC algorithm for implementation after simplification is shown in Figure 3.3, while the required signal processing of the DAGC unit after simplification is presented in Figures 3.4.

Figure 3.3. Schematic DAGC Algorithm Diagram after Simplification

- $|X_i| = \_\_abs(|X_i|)$
- $int(ln|X_i| / ln2) = 31 - \_lmbd()$
- Let $int(a \times \sum_1^{384} int(ln|X_i| / ln2)) = -coeff$
- $f \times X_i = 2^{int(a \sum_1^{384}(..))} \times X_i = (X_i >> coeff)$

  by realizing that -coeff is negative

Figure 3.4. DAGC Implementation after Simplifying Algorithm



Comparing Figures 3.3 and 3.4 to Figures 3.1 and 3.2, respectively, it demonstrates that after the simplification, the originally seemed-to-be complicated processing turns out to have a rather simple solution. This is quite amazing. The followings summarize the major steps for the simplifying procedures.

- $\ln|X_i|/\ln 2$ is equivalent to $\log_2|X_i|$. Taking the integer value of $\log_2|X_i|$ is equivalent to finding the left most bit ONE in $|X_i|$. For example, if $|X_i| = 2$, then $\log_2|X_i| = 1$; if $|X_i| = 3$, then $\log_2|X_i|$ is approximately equal to 1.5850. Taking the integer value of $\log_2|X_i|$ in both cases yields an output of 1. Following the standard practice and calling the right most bit of a 32-bit word Bit #0, then what $\text{int}(\log_2|X_i|)$ actually produces is the bit number of the left most bit ONE in $|X_i|$.
- Another C intrinsic function _lmbd() is used in the program to simplify this bit searching process. Function _lmbd() searches for the leftmost 1 or 0 and returns the number of bits up to the bit change. For example, _lmbd(1, 2) = 30 (yielding the number of 0's up to the first 1) whereas _lmbd(0,2) = 0 (yielding the number of 1's up to the first 0). Thus, $\text{int}(\ln|X_i|/\ln 2)$ can be alternatively implemented as 31 - _lmbd(1, input). This alternative step changes a rough floating point movement into a very simple fixed point calculation. So the most difficult challenge, huge amount of floating point calculations, is successfully solved.
- Going further along Figure 3.2, the blocks that calculate the value of 2^w and multiply it with X_i can also be implemented in a much simpler way. With w denotes the integer value of $a \times \sum_{i=1}^{384} \text{int}(\ln|X_i|/\ln 2)$, S_i's can be obtained by

shifting X_i's to the right by –w number of bits. This understanding is reached by first noting that w is a negative integer; then multiplying X_i with 2^w is equivalent to dividing X_i by 2^{-w} or $2^{|w|}$.

- In fixed point DSP, any division by a power of 2 integer can be achieved by a simple right shifting. Of course, there is a slight difference in the actual implementation depending on whether the input symbol is positive or negative. If the input symbol is a positive number, then the right shifting is all that is needed for the division. If the input symbol is a negative number, then one needs to be added to the result of the right shifting to compensate for the sign extension due to the bit shifting.
- Floating point calculations have been substantially minimized as shown in Figures 3.3 and 3.4, and discussed above.

3.2.5 Summary

Various techniques have been taken to successfully implement the DAGC unit on C6201 in C program. The C intrinsic functions are employed to improve the efficiency, and the algorithm is simplified through effective conversions. This series of conversions make the DAGC unit readily implemented on the C6201 DSP, and substantially minimize the floating point calculations, greatly increase the processing speed and reduce the error range. The benchmark result for the DAGC function to process one frame of data (384 symbols) is 4116 CPU cycles, which is about 0.10% of the processor time. This is a very impressive and satisfactory result for the DAGC implementation.

CHAPTER 4: IMPLEMENTING VITERBI DECODER ON C6201

The Viterbi Decoder is the last major component of the demodulation process to be implemented for this project. It is also the most computationally demanding component of the entire demodulation process. The Viterbi Algorithm (VA) is a very well studied subject. There are already many established techniques for implementing the VA. These techniques naturally form the corner stones of this specific implementation. However, there are more techniques to be explored here that are aimed specifically to take the advantage of the C6201's architecture and its C compiler. Those C6201 specific techniques are of great importance for making the Viterbi Decoder to meet the real time constraint imposed by the IS-95 system. Without them, it would be quite difficult for the implemented program to accomplish the task in pure C; besides, the efficiency of the implemented program would also be substantially reduced.

4.1 The Convolutional Encoder

Viterbi Decoder is also the most complicated component to understand among the entire demodulation process. Its operation is intimately related to its counterpart at the transmitter's end: the convolutional encoder. Therefore, understanding the convolutional encoder is vital for implementing the Viterbi Decoder efficiently.

For the forward traffic channel generation, the IS-95 Standard specifies that the information bits be convolutionally encoded. Convolutional coding provides redundancy that the receiver uses to correct errors due to transmission distortions.

The VA is a maximum likelihood (ML) decoder. Viterbi specifically indicates the use of VA as the optimal decoder for convolutionally encoded data [Viterbi, 1995].

The IS-95 Standard also specifies the use of Viterbi Decoder for the demodulation of the Forward CDMA channel.

Both cellular and PCS band can use either rate set 1 or rate set 2 vocoder. The IS-95 Standard specifies the use of a rate ½, constraint length (K) 9 convolutional encoder for rate set 1 vocoder. The rate ½ means that the encoder produces two coded bits as output for every input information bit. Rate set 2 has a ½ rate encoder followed by puncturing to produce an effective coding rate of ¾. In both cases, a constant symbol rate of 19.2 ksps is maintained. The constraint length indicates how many delayed elements will be used in generating the current outputs. For example, for a rate ½, and K = 9 encoder, the current information bit along with eight most recent uncoded information bits would be used in producing the two current coded bits. In the actual implementation, when the encoder is implemented as a shift register, this would involve using eight delay elements to keep the past information bits in the memory. The following figure gives an illustration of such an encoder:

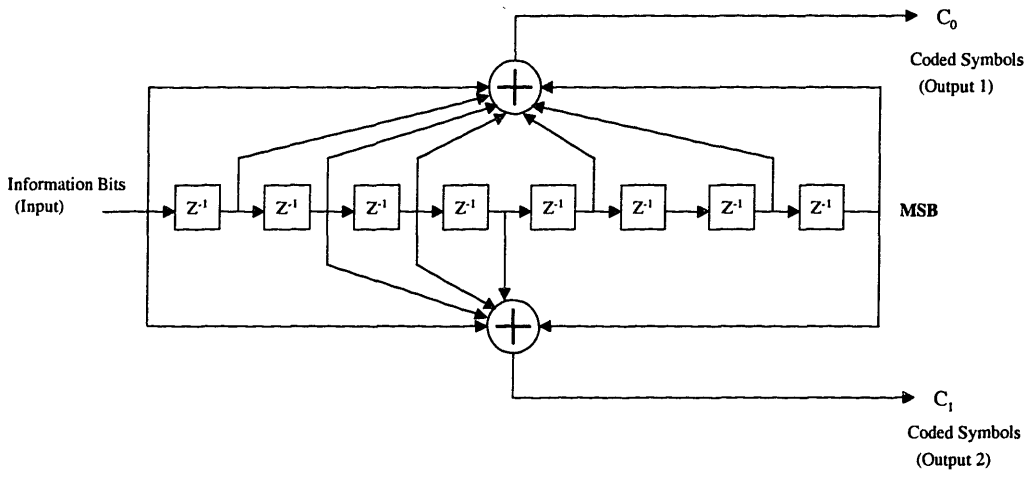


Figure 4.1. Convolutional Encoding, Rate ½, K = 9
(Figure 3-6 of IS-95A: The CDMA Standard on p.3-10)

Higher constraint length provides more coding gain. However, complexity increases exponentially with constraint length. Increasing K beyond 9 would increase the coding gain slightly with a great increase in complexity [Qualcomm, 1997]. The current state of the art limits decoders to a constraint length of about K = 10 [Skylar, 1988]. In "Digital Communications", Sklar [1988] discussed the details on comparison of coding gains for different constraint length.

The upper and lower branch connection points of the K = 9 shift register in Figure 4.1 can be described by the following two polynomials:

$$C_0(x) = 1 + x + x^2 + x^3 + x^5 + x^7 + x^8 \qquad \text{(Eq. 4.1)}$$

$$C_1(x) = 1 + x^2 + x^3 + x^4 + x^8 \qquad \text{(Eq. 4.2)}$$

The polynomial generators of a convolutional code are usually selected based on the code's free distance properties. Sklar [1988] also presented a comprehensive discussion of the related criteria in his "Digital Communications". The IS-95 Standard has chosen the above two polynomials because they offer the optimal Euclidean distance for a rate ½ encoder. The Euclidean distance represents a measure of the degrees of orthogonality among possible sequences. Selecting polynomials with the highest degree of orthogonality or optimal Euclidean distance maximizes the probability of correct detection at the receiver's end. Eq. 4.1 denotes the upper connection of the shift register leading to coded bit C_0 whereas Eq. 4.2 is a representation of the lower branch connection leading to C_1. The coefficients of the code polynomials can be conveniently represented as octal 753 and 561. A table listing of the polynomial coefficients has been provided in Digital Communications [Proakis, 1995].

4.2 General Process for Implementing Viterbi Decoder

Numerous past works on implementing the Viterbi Decoder have contributed to a commonly agreed procedure for implementing the Viterbi Algorithm (VA). Figure 4.2 illustrates the general process for implementing a soft decision (Euclidean) Viterbi Decoder. Generally, soft decision Viterbi outperforms hard decision (Hamming) Viterbi since it takes into account the relative uncertainty level of the data. Therefore, it is of higher interest to consider the soft decision VA here.

The general decoding procedure consists of the Add-Compare-Select (ACS) operation and the Traceback operation. The ACS is actually a path selection and metric accumulation operation. The path with the highest accumulated metric is evaluated as the most possible sequence. Convolutionally encoded data is decoded through knowledge of the possible state transitions. The VA efficiently limits the number of possible paths for consideration.

- The VA notes that for a rate ½ encoder, there are only two possible encoder states at Stage 1 that can enter into a particular encoder state at Stage 2; and there are only two possible encoder states at Stage 3 that any encoder in Stage 2 can enter. Here, Stages are references to the time frame.

- If two nodes are merging into the same node, then only one of them needs to be kept since their path after the merging would be indistinguishable. The VA adds the new metrics (local distances) to the accumulated metrics associated with the nodes of each trellis stage, select the one with the higher accumulated metric (more likely path sequence), and stores the decision of which path it has chosen.

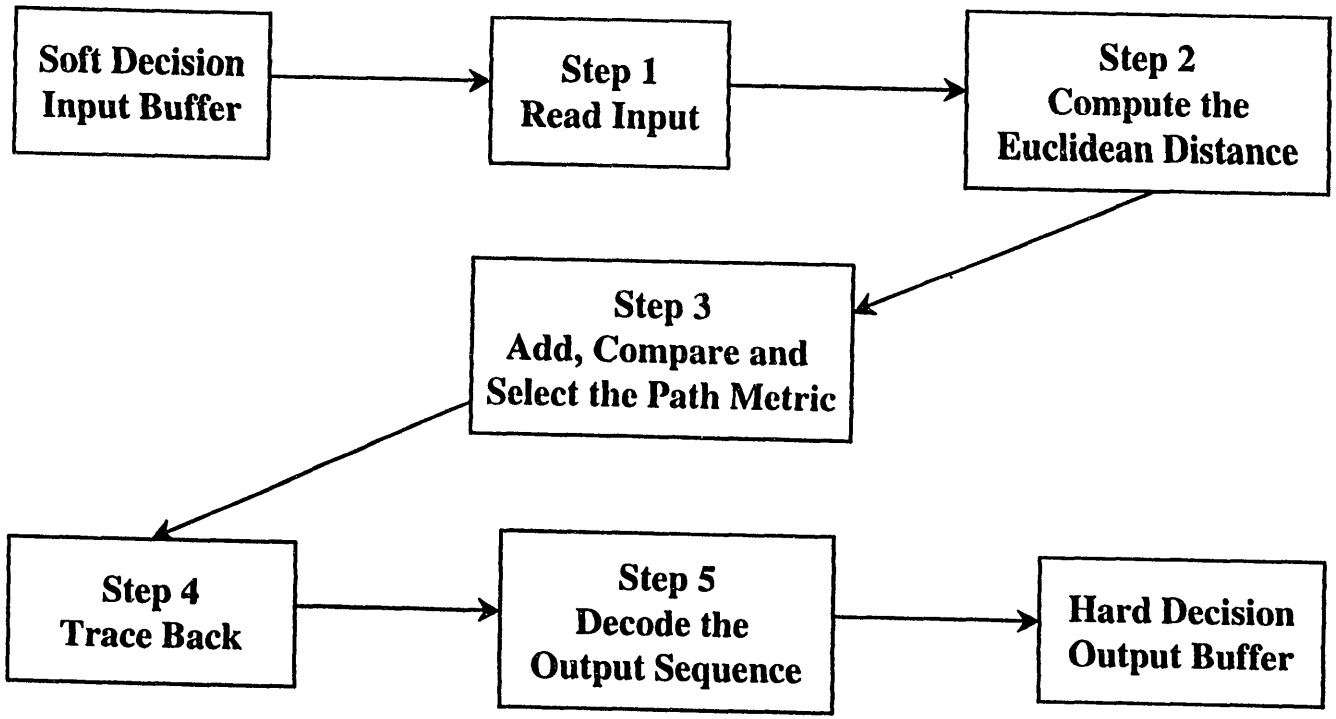


Figure 4.2. General process flow chart for soft decision Viterbi Decoder

- After reaching the end of the input sequences, the VA selects the node with the highest accumulated metric and traces its way back through the trellis according to the path decision memory it has stored. The traceback path formed would consist of different states along the trellis. However, the trellis stage states' numbers are exactly consisted of the uncoded information bits that the decoder is trying to get.

4.3 Implementing IS-95 Viterbi Decoder on C6201

VA is the most computationally intensive part of the demodulation process. To implement Viterbi Decoder efficiently is vital for meeting real time constraint. The Viterbi Decoder has been implemented for a K = 9, rate ½ convolutional encoder. This is a soft decision VA specifically designed for the transmission of a full rate data frame. Figure 4.3 is the flow chart for implementing the VA.

Figure 4.3 follows the general flow process of Figure 4.2. However, there are some important procedural differences employed in the actual implementation here that is worth noting:

- In this IS-95 implementation, the traceback function is implemented over five times the constraint length rather than over the entire data frame to minimize the delay and memory storage required by the decoding process. As a consequence, two types of traceback function are created for dealing with the situation. The function traceback1() does limited traceback and decodes a small number of bits while the VA still reads input sequences. The function traceback2() is employed for decoding all the remaining bits once the entire input data set has been read.

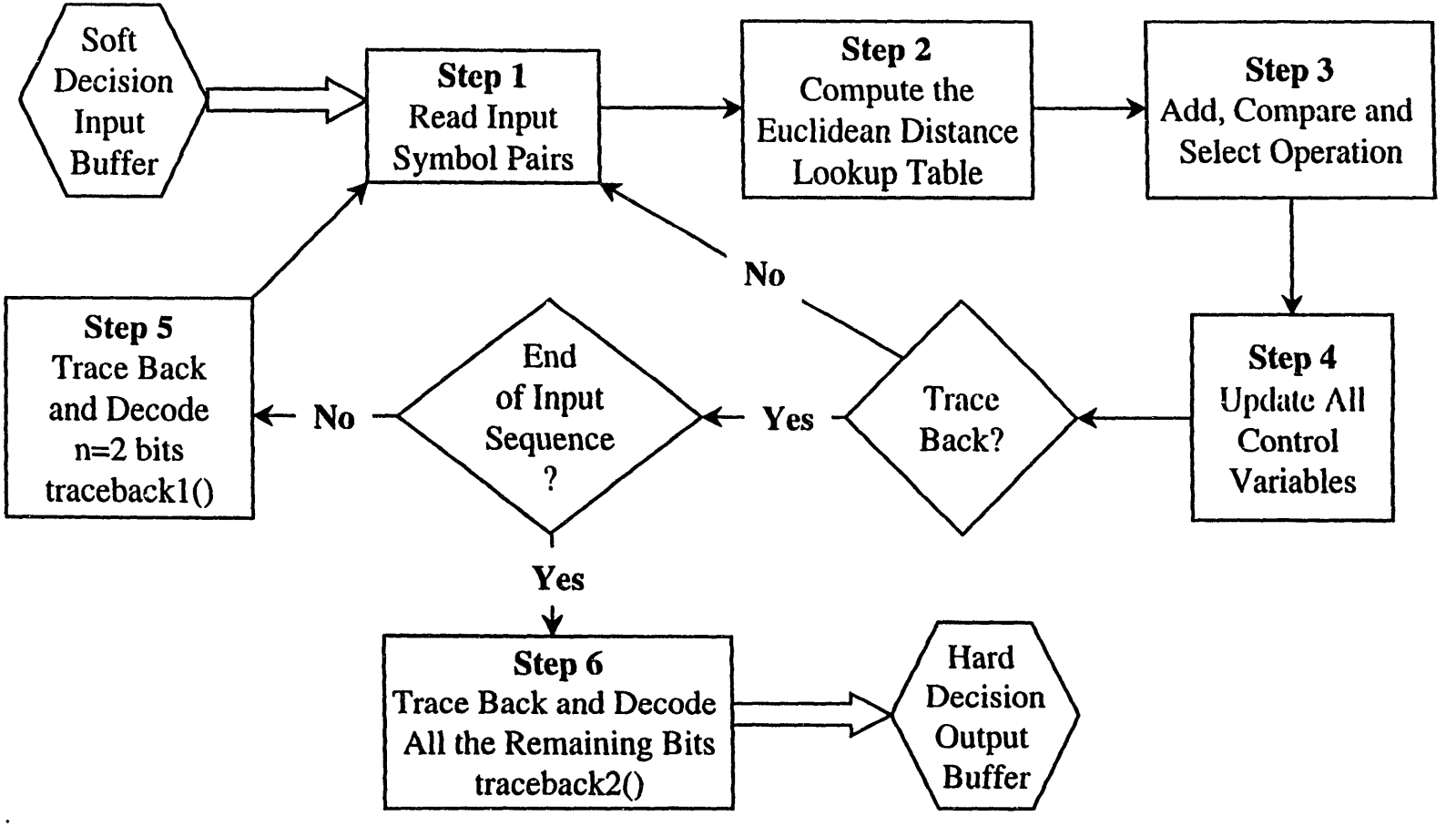


Figure 4.3. Viterbi Decoder for rate 1/2, K = 9 convolutional encoder

- Since a maximum-likelihood (ML) sequence means the most likely sequence over the entire data set, VA of this particular implementation is theoretically not a truly maximum-likelihood decoder because the decoded bit is only based on less than one fourth of the available data at any point of the decoding process. However, in practice, the data sequence converge in less than five times the constraint length, so there is little performance sacrificed in becoming a slightly sub-optimal decoder.

Besides those points made above, there are many other important techniques employed at each step of the implementation that contribute greatly to the algorithm's impressive efficiency. In the following paragraphs, the major techniques used will be discussed with their applications associating to each step of the process.

Step 1:

Step 1 of the flow chart is to read the input symbol pairs. There are 384 symbols for each frame of data. For this full rate implementation, two symbols are read for each trellis stage. So a total of 192 trellis stages need to be computed. Each trellis stage is composed of many nodes (delay states) with its amount depending on the constraint length of the convolutional encoder. A K = 9 convolutional encoder has 8 delay elements. So there are $2^{9-1} = 256$ delay states in each trellis stage.

When implementing on the C6x, it is important to choose just the right word length for data. The C6x can do twice the loads, stores, and additions on each function unit per cycle if the data is of type short integer rather than integer. After the DAGC, it is possible to represent the input symbols to the Viterbi Decoder as a short integer instead of an integer. The usage of appropriate word length has helped to improve the final benchmark results slightly. When initially the inputs to the Viterbi Decoder are 32-bit

integers, it takes 919 cycles per trellis stage. After making the inputs to 5-bit data, which only requires short integer for representation, the cycle counts for each trellis goes down to 889 cycles.

Prelude of Step 3:

Step 3 does the Add-Compare-Select operation. The ACS operation is generally performed between two nodes of the trellis in a butterfly. This is most easily seen in the following diagram.

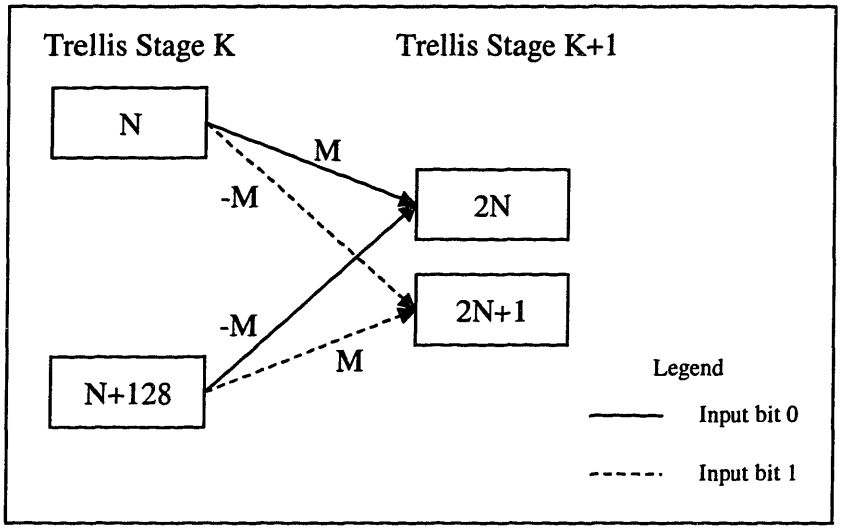


Figure 4.4. Trellis Butterfly Diagram, Rate ½, K = 9

Nodes N and N+128 of Trellis Stage K both can potentially enter Nodes 2N and 2N+1 of the next trellis stage. Both Nodes N and N+128 have in their record the accumulated path metrics up to Stage K. There are always two transition possibilities for each node depending on whether the input bit is 0 or 1. The ACS operation attempts to add the new local distance to the old metrics, select the one with the higher metrics, and store the new metric with the associated nodes for the next trellis stage.

Quite often, this ACS operation is computed as a butterfly function. The new local distance is calculated based on the input symbols, the value is passed as a parameter to the function bufferfly(); the ACS operation is performed within the function call. For an IS-95 system, this would mean calling the bufferfly() function 128 times for each trellis stage update.

However, through trials of experimentation, it is discovered that the conventional way of implementing the ACS as an individual function is not efficient when implemented on the C6x DSP. C6201 relies heavily on the processor pipelining. Making function calls breaks up the pipelining of the processor. Therefore, it is much better to implement the ACS in a for-loop as to keep the smooth flowing of the pipeline.

This naturally brings the need of pre-calculating and ordering the local distances for convenient indexing of the ACS operation in a loop. Step 2 of the Flow Chart has been invented specifically for this purpose.

Step 2:

Step 2 computes the Euclidean Distance Lookup Table. Rather than calculating two local distances for each butterfly, which in turn would require calculating the local distance 256 times for each trellis stage, this step is greatly expedited by developing some very important improvements as discussed below.

Generally, the soft decision local distance could be calculated according to the following equation [Hendrix, 1996]:

$$\text{Local Distance} = SD_0 * C_0(j) + SD_1 * C_1(j) \qquad \text{(Eq. 4.3)}$$

Where SD_0 and SD_1 are the two input symbols to the Viterbi Decoder; $C_0(j)$ and $C_1(j)$ are the two coded output of the rate ½ convolutional encoder as shown in Figure 4.1.

There are a few interesting discoveries being made about the components of Eq. 4.3:

- First of all, each state transition yields two bits as its output. Therefore, there are only four possible choices for the $C_0(j)C_1(j)$ combination: 00, 01, 10, and 11.
- $C_0(j)$ and $C_1(j)$ are binary bits as demonstrated in Figure 4.1. However, they should be considered in their antipodal form in Eq. 4.3, i.e., 0's represent +1's and 1's represent –1's. Then the local distance will be a sum of two numbers: sign SD_0 + sign SD_1.
- Since the Euclidean local distance is calculated based on Eq. 4.3, there are only four possible combinations of local distance that can be produced by any state transition, i.e., $SD_0 + SD_1$, $SD_0 - SD_1$, $-SD_0 + SD_1$, $-SD_0 - SD_1$.
- Further, according to Figure 4.4, the local distances (M) involved for each pair of butterfly are the negatives of each other, i.e. $SD_0 + SD_1$ and $-SD_0 - SD_1$ occur in pairs, $SD_0 - SD_1$ and $-SD_0 + SD_1$ occur in pairs.
- All the 128 butterflies within one trellis stage use one of the two pairs of local distances calculated in the previous step.
- Butterfly ACS computation has a fixed structure as presented in Figure 4.4. Therefore, what is needed is just to determine the M values of the butterflies and store them in appropriate sequence for use by the ACS operation.

These discoveries bring a great simplification to the problem. Now, the real work for calculating the local distance for all the 256 nodes is to find out the coded bits that are produced for each state transition. In addition, another advantage achieved here is that this job does not need to be done real time and could be pre-calculated and programmed. The relevant coded bits information for a rate ½, K = 9 encoder are calculated and summarized in the following table:

Coded bits yield by State Transition	**Butterfly # = N (States # = N, N+128)**
00 (Input bit = 0) or 11 (Input bit = 1)	0, 6, 11, 13, 17, 23, 26, 28, 32, 38, 43, 45, 49, 55, 58, 60, 65, 71, 74, 76, 80, 86, 91, 93, 97, 103, 106, 108, 112, 118, 123, 125.
01 (Input bit = 0) or 10 (Input bit = 1)	3, 5, 8, 14, 1, 18, 20, 25, 31, 35, 37, 40, 46, 50, 52, 57, 63, 66, 68, 73, 79, 83, 85, 88, 94, 98, 100, 105, 111, 115, 117, 120, 126.
10 (Input bit = 0) or 01 (Input bit = 1)	1, 7, 10, 12, 16, 22, 27, 29, 33, 39, 42, 44, 48, 54, 59, 61, 64, 70, 75, 77, 81, 87, 90, 92, 96, 102, 107, 109, 113, 119, 122, 124.
11 (Input bit = 0) or 00 (Input bit = 1)	2, 4, 9, 15, 19, 21, 24, 30, 34, 36, 41, 47, 51, 53, 56, 62, 67, 69, 72, 78, 82, 84, 89, 95, 99, 101, 104, 110, 114, 116, 121, 127.

Table 4.1. Lookup Table for States Transition, Rate ½, K = 9

Figure 4.4 and Table 4.1 together tell quite a bit about how a rate ½, K = 9 decoder could be constructed. For example, if the current state is 100, then N = 100. Figure 4.4 says that the next state would be 200 if the current input bit is 0 and the next state would be 201 if the current input bit were 1. Following the convention, the right most bit in the convolutional shift register is considered to be the most significant bit of the state.

Function branchmetric() is responsible for both reading the input symbols and calculating the local Euclidean distance. Function branchmetric() first calculates the four possible values of the local distance, then sets up the look up table according to the information provided by Table 4.1.

Step 3:

Step 3 does the Add-Compare-Select operation. There are 128 ACS operations performed for each stage of memory update. With a symbol rate of 19.2 Ksps, the ACS operations are computed 24,576 times for each frame (20 msec) of data. And this is obviously a large amount of calculation.

The butterfly computation is a highly parallel operation. This means that the instructions would have a constant demand for the same type of function units. But it also means that two butterflies can be computed independent of each other's result. The two notions make pipelining especially important. As discussed previously, the computation of the ACS operation in this program has been implemented as a for-loop to expedite pipelining. The entire loop is in charge of one trellis stage. The inner loop calculates the ACS for sixteen butterflies. It stores the path chosen by all the 32 nodes in a single 32-bit word (two nodes form one butterfly).

Even though only a single 32-bit word of storage is needed to satisfy the memory storage of the entire inner loop, two temporary 32-bit words have been used within the inner loop. This is an important cycle-saving technique used here within the inner loop for recording to avoid the delay of two writes to the same destination. The C6x processor allows four memory accesses to the same register each cycle, which includes three reads and only one write. Two writes to memory are sometimes needed when computing one

butterfly. If only one register is allocated for writing, one cycle of delay could be caused when the second write is needed. On the other hand, allocation of two variables for recording allow both writes to memory be completed within one cycle. The result of the two 32-bit words is then combined to a single word at the end of the sixteenth butterfly. This technique of creating a second temporary storage for path recording yields significant improvement over the benchmark result.

The inner loop is then repeated by the outer loop for 16 times to capture all the 256 nodes of each trellis stage. The recording of path bits require the usage of sixteen 32-bit words for each trellis stage.

An application of similar spirit is made on the accumulation metrics. Both array cmetric[256] and nmetric[256] are used for metric accumulation. For odd number of trellis stages, cmetric[256] is the old array, and nmetric[256] = cmetric[256] + local distance is the new array. For even number of trellis stages, nmetric[256] is the old array and cmetric[256] is the new array. The use of dual metric accumulation arrays simplifies the programming so that the memory sets would not have to be copied over and over. As seen from the program itself, each outer loop does the calculations for two trellis stages.

Step 4:

Step 4 is a rather simple part of the decoder program that updates all the control variables.

Step 5:

Step 5 does the decoding before the end of the input sequence has been reached. Function traceback1() has been used to achieve this task. The purpose for this implementation is primarily due to the memory storage concern. Each trellis stage

requires sixteen 32-bit words as storage. This would require 3072 32-bit words for storage if decoding were done only once after processing all the input symbols. That is too much memory to be occupied on a DSP. Thus, an important alternative has been developed to avoid this problem in an effort to improve the efficiency. That is to do decoding before reaching the end of the input sequence and just after enough data has been accumulated for decoding. The data typically converge in less than 5 times the constraint length. Fifty (50) trellis stages have been chosen as the storage path in this research. Higher storage path enables more decoding bits at each traceback, but it is achieved at the expense of consuming more memory for storage. In this implementation, it has been chosen to decode 2 bits at each traceback1(). However, it is believed that more bits could be decoded fore each traceback1().

Step 6:

Step 6 does the traceback for the remaining bits once all the inputs have been processed. This is implemented by function traceback2(). At the encoder's end, eight 0 bits are inserted at the end of each data frame for returning the encoder back to its zero state. The zero insertion is extremely important for decoding. This information assures that each traceback2() function has the zero state as its initial state. Since the starting point for traceback2() is certain, and its ending point is converged, all the remaining bits could be decoded at once.

C6x depends heavily on the optimization level of the code. Thus, it is very important to be a smart programmer when programming for C6201 in C. The following techniques have been developed during the implementation process of the Viterbi

Decoder. They are very valuable in helping the compiler to compile efficiently, which in turn helps the C program to run faster.

1) Changing butterfly() function calls to the for-loops

Using the for-loops to substitute the butterfly() function calls in the Add-Compare-Select operation is one of the important steps developed for this program to improve efficiency. It is extremely important that Add-Compare-Select operation has been implemented in these for-loops. The initial implementation involved implementing the Add-Compare-Select operation for a butterfly as a function (this basically means that updating each trellis stage involves making 128 calls to this butterfly() function). However, making function calls like this breaks up the pipelining of the process. Thus, it is very important to avoid doing such kind of operations. After changing butterfly() function calls to the for-loops that are present in the program now, the benchmark is improved tremendously.

2) Skills on good indexing of the arrays

The second important thing to note is that the calculation of Viterbi involves lots of indexing of the arrays. What is often neglected is the emphasis on the correct indexing of these arrays. The correct strategy to do the job is to give the compiler as much information about where the data is coming from as the programmer can possibly do. This greatly helps improving the pipelining as well. Note the way the indexing has been done in the Viterbi program, it gives the compiler the full knowledge of each symbol's relative position to others. Do not give the compiler a black box, i.e., thinking that 16*a+k is used repeatedly and replace it with some variable x for indexing. Doing this breaks up the pipelining and slows down the processor substantially.

3) Keeping the pipelining flowing

The extremely important thing to note for programming in C6x is that it is a highly paralleled processor. It has eight function units and is very important to get as many function units running as that is possible for the algorithm. Therefore, keeping the pipelining flowing is so important that it can never be over emphasized.

4) Avoid using the standard C library function calls

Avoid using the standard C library function calls. It is convenient for regular programming to get something to work but is too slow for DSP implementation. Function calls generally break up the pipelining. Thus, the usage of function calls should be minimized.

5) Avoid using % for indexing

Avoid using such operations like % for indexing. The current hardware does not support it well. Indexing using % operation also requires a hidden function call to somewhere else. It is fine if this operation is only used for a limited amount of time; if it is in a loop and is used repeatedly, however, it would be something to be watched out.

6) Keep program simple

Keep one's program simple. It is generally the best way to improve the compiler efficiency. Only if the work can be done, the program should be kept as simple as possible.

4.4 Viterbi Decoder Benchmark Result on C6201

The Viterbi Decoder is implemented entirely in C. The benchmark for decoding one frame of data (384 symbols) is 368.6 K cycles. This is about 9.22% of the CPU time. The Add-Compare-Select operation takes 889 cycles per trellis stage including overhead

with a 4-cycle kernel per butterfly. This result compares very well to TI-internal hand scheduled assembly performance for the same type of Viterbi Decoder which also has a 4-cycle kernel per butterfly → a benchmark result of 512 cycles + overhead per trellis stage. The Compiler flag used for benchmarking is –g –o3 –k –mg.

This code contains some fscanf(), and fprintf() functions. These are for reading inputs from an input PC file and send the outputs of Viterbi to a PC file. The entire purpose of their presence is to check the accuracy of the Viterbi Decoder. In real time processing, inputs to Viterbi would not be taken from a PC file, so when benchmarking the Viterbi code, these file I/O operations should be commented out. Otherwise, PC file I/O operations go through JTAG and the Go DSP profiler can not clock the CPU cycle counts accurately.

4.5 Constructing VA for Half, Quarter and Eighth Rate Encoders

The IS-95 system supports a variable rate vocoder. The decoder has no information on which rate has been transmitted. Therefore, the demodulation process needs to allow the decoding at all four possible rates; then compare the CRC bit of the decoded sequence to decide on the actual sequence that has been transmitted.

The operations for constructing the Viterbi Decoder for the half, quarter, and eighth rate vocoder are essentially the same as for the full rate vocoder. The only difference lies in that the input symbols to Viterbi need to be combined before being fed into the Viterbi. Figures 4.5 through 4.7 illustrate the idea.

For decoding of the quarter and eighth rate vocoder, the number of input symbols to Viterbi is decreased to 96 and 48 symbols respectively; this causes the number of trellis stages to decrease to 48 and 24 stages respectively. In these cases, memory storage

of the chosen data path is no longer an issue. The entire data sequence could be decoded once, eliminating the use of function traceback1(). This idea is illustrated in Figures 4.6 and 4.7.

From the illustration of the decoding for other vocoder rates, it is possible to get an approximate benchmark for the number of cycles needed to decode one frame of data. The decoding of the half, quarter, eighth rate vocoder should be strictly less than half, quarter, and eighth of the total cycle counts for decoding the full rate vocoder, respectively. The reason is that the number of ACS operations needed is proportional to its rate. However, the decoding for lower variable rate vocoder decreases or eliminates the need for using function traceback1(); thus it should result in a lower cycle count.

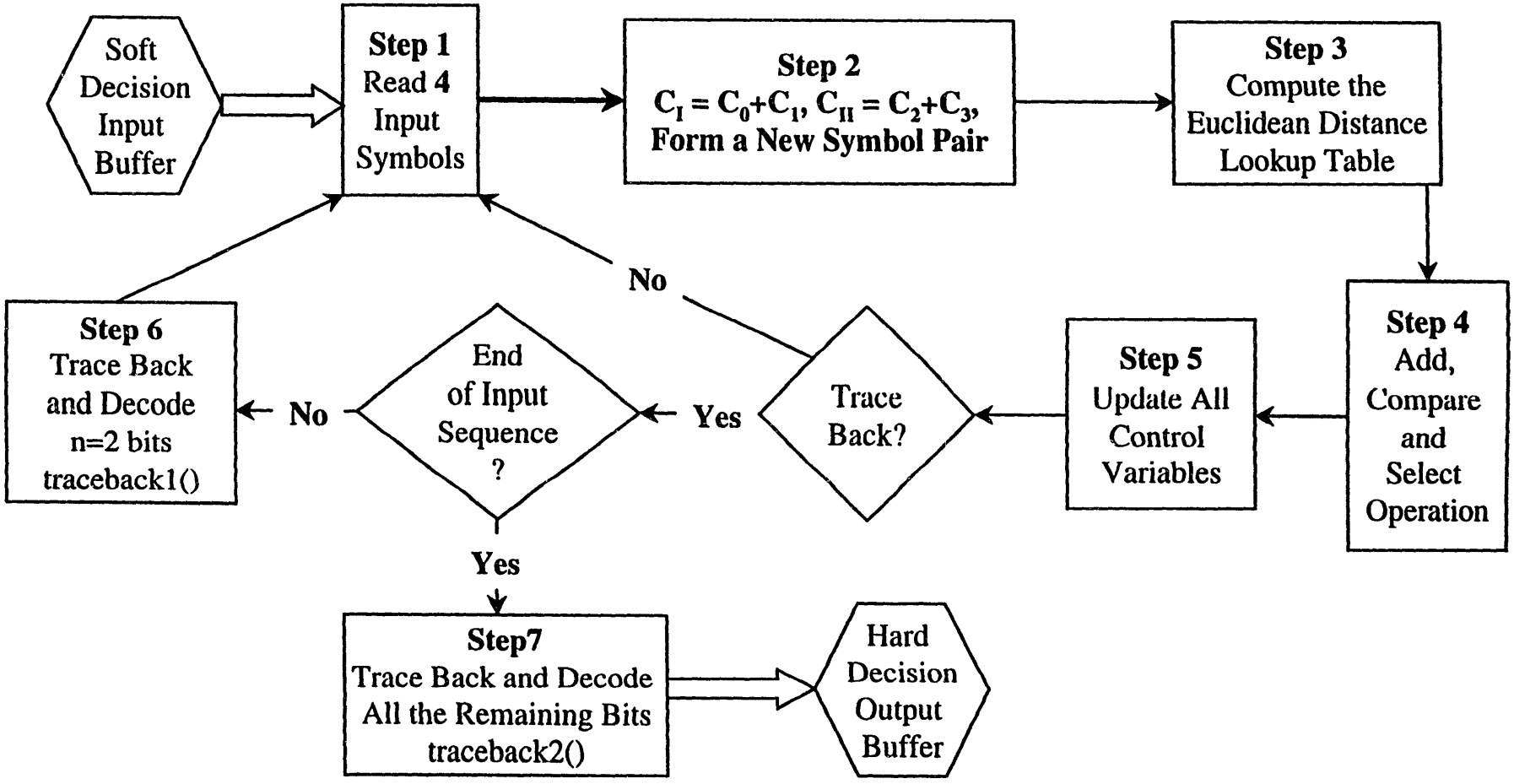


Figure 4.5. Flow chart for soft decision Viterbi Decoder: Rate Set 1, Half Rate Frame

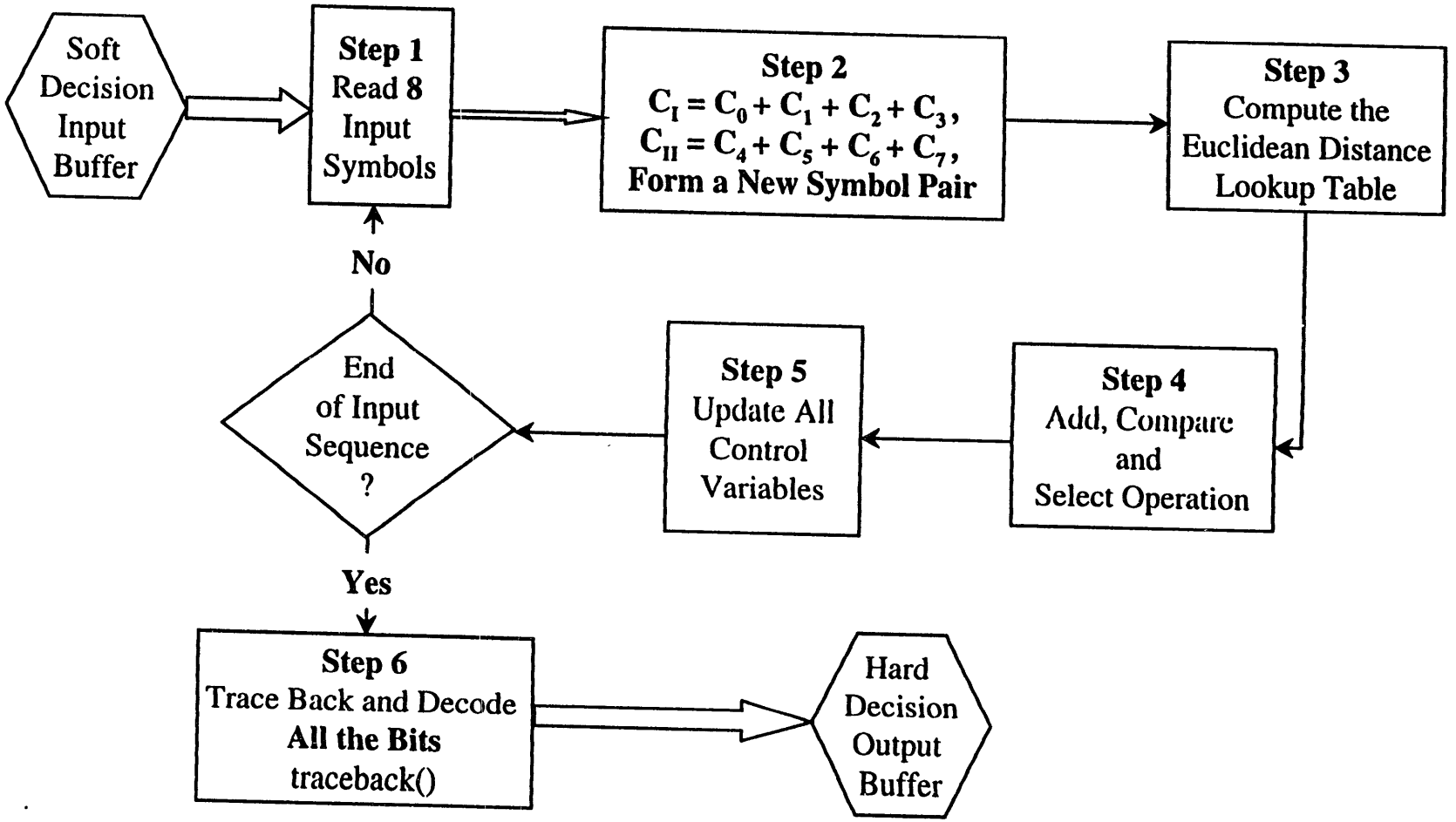


Figure 4.6. Flow chart for soft decision Viterbi Decoder: Rate Set 1, Quarter Rate Frame

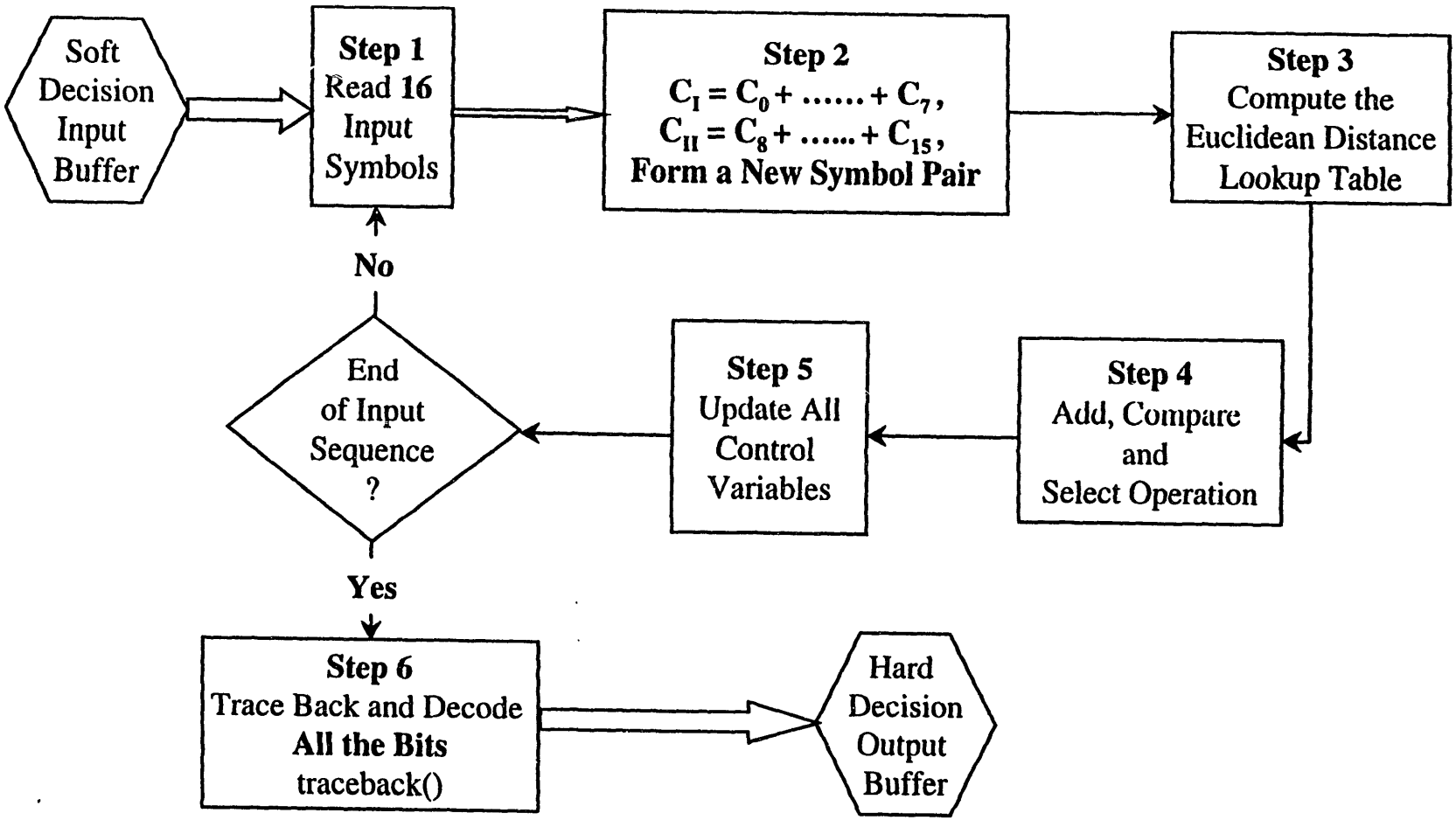


Figure 4.7. Flow chart for soft decision Viterbi Decoder: Rate Set 1, Eighth Rate Frame

4.6 Summary

The Viterbi Decoder has been successfully implemented on C6201 entirely in C. A lot of improvements have been developed in this project that are aimed specially to take the advantage of the C6201's architecture and its C compiler in order to greatly raise the efficiency and meet the real time constraint imposed by the IS-95 system.

The benchmark for decoding one frame of data (384 symbols) is 368.6 K cycles. This is about 9.22% of the CPU time. The Add-Compare-Select operation takes 889 cycles per trellis stage including overhead with a 4-cycle kernel per butterfly. The benchmark result compares very well to TI-internal hand scheduled assembly performance for the same type of Viterbi Decoder. The Compiler flag used for benchmarking is –g –o3 –k –mg.

CHAPTER 5. IMPLEMENTING PN ON C6201

5.1 Pseudorandom Noise Descrambling

Following the orthogonal spreading, each code channel is spread in quadrature. The spreading sequence, called the pilot Pseudorandom Noise (PN) sequence, is a quadrature sequence of length 2^{15} (i.e., 32,768 PN chips in length). The spreading sequence is based on the following characteristic polynomials:

$P_I(x) = x^{15} + x^{13} + x^9 + x^8 + x^7 + x^5 + 1$ for the in-phase (I) sequence

and

$P_Q(x) = x^{15} + x^{12} + x^{11} + x^{10} + x^6 + x^5 + x^4 + x^3 + 1$ for the quadrature (Q) phase sequence (TIA/EIA/IS-95-A, p. 7-18)

The C code that generates the PN sequences was provided by Dr. Aris Papasakellariou of Texas Instruments. It is included here as the completeness to the project. The PN generator implements two maximum length linear feedback shift register sequence {i(n)} and {q(n)} based on the above polynomials. They are of length $2^{15} - 1$ and is generated by the following linear recursions:

$i(n) = i(n-15) \oplus i(n-10) \oplus i(n-8) \oplus i(n-7) \oplus i(n-6) \oplus i(n-2)$

(based on $P_I(x)$ as the characteristic polynomial)

and

$q(n) = q(n-15) \oplus q(n-12) \oplus q(n-11) \oplus q(n-10) \oplus q(n-9) \oplus q(n-5) \oplus q(n-4) \oplus q(n-3)$

(based on $P_Q(x)$ as the characteristic polynomial)

where i(n) and q(n) are binary-valued ('0' and '1') and the additions are modulo-2 [TIA/EIA/IS-95-A, p. 7-18].

5.2 Summary

The PN generator in its current form takes approximately 1.2 million CPU cycles to generate one frame of the I and Q PN sequences (24,576 chips for each sequence); that is about 30% of the CPU time. It takes approximately another 51,000 CPU cycles for the demodulation, which is nearly 1.28% of the CPU time.

The bottleneck of the PN Descrambling function lies in generating the spreading quadrature sequences. The current PN generator is very inefficient in this aspect. Even though the C6201 is still quite capable of meeting the real time constraint with this inefficiency, better alternatives should be investigated for the future work. Either a simplification of the algorithm is needed, or implementing the sequence generation function in special hardware might indeed be a better solution.

CHAPTER 6. CONCLUSIONS

The IS-95 system, the industrial standard for CDMA, has been successfully implemented in software on TI fixed-point DSP TMS320C6201, and met the real time constraint. The project includes all the major components of the demodulation process for the forward link system: PN Despreading, Walsh Despreading, Phase Correction & Maximal Ratio Combining, Deinterleaver, Digital Automatic Gain Control, and Viterbi Decoder. The entire demodulation process is done completely in C.

Intensive efforts have been concentrated on developing various specific techniques to optimize the design for all the components involved during the whole implementation process. These developments are accomplished by making the best use of their unique characteristics to simplify the algorithms of concern, taking the advantages of the C6201's architecture, and its C Compiler. Simplifying the algorithms is one of the keys on implementing the IS-95 system on C6201 DSPs. This principal has been applied in the entire implementing process. The algorithms of the different components of interest have been analyzed and studied thoroughly. Some very interesting phenomena have been observed, and efficiently used in the programming process.

Overall, the major specific techniques employed in the project include but not limit to the followings: to simplify the algorithms first before programming, to look for regularity in the problem, to work toward the Compiler's full efficiency, and to use C intrinsics whenever possible. All these attributes together make the implementation scheme great for DSP applications. The benchmark results compare very well to TI-internal hand scheduled assembly performance for the same type. To be specific, when

using -g -o3 -k - mg compiler flag for all benchmark, the benchmark result over one frame for the different components are as follows:

- Walsh Despreading = 63.6 K CPU cycles;
- Phase Correction/MRC = 25.4 K;
- Deinterleaver = 1.4 K;
- DAGC = 4.1 K;
- Full Rate Viterbi = 368.6 K;
- Total CPU cycles available = 4,000 K; and
- Estimated Percentage of Usage = 21.18% (including all 4 rates of Viterbi but excluding PN descrambling)

The overall process time-shares of all the components are presented in Figure 6.1.

This successful implementation provides a very attractive alternative for the future applications of DSPs. It is well known that ASIC design is expensive and time consuming, programming in assembly is easier and cheaper, but programming in C is a much easier and efficient way out, in particular, for general computer engineers. Currently, the 2nd generation CDMA phones are produced by Qualcomm. Texas Instruments (TI) has ASIC design for Viterbi decoder on C54x. Several of the components in the forward link process are also implemented in hardware. However, having to design a specific hardware for a particular application is not only expensive but also time consuming. Thus, the possibility of the alternative implementations has been of great interest to both customers and TI itself. The successful implementation of IS-95 entirely in C on TI C-6201 provides such kind of alternatives.

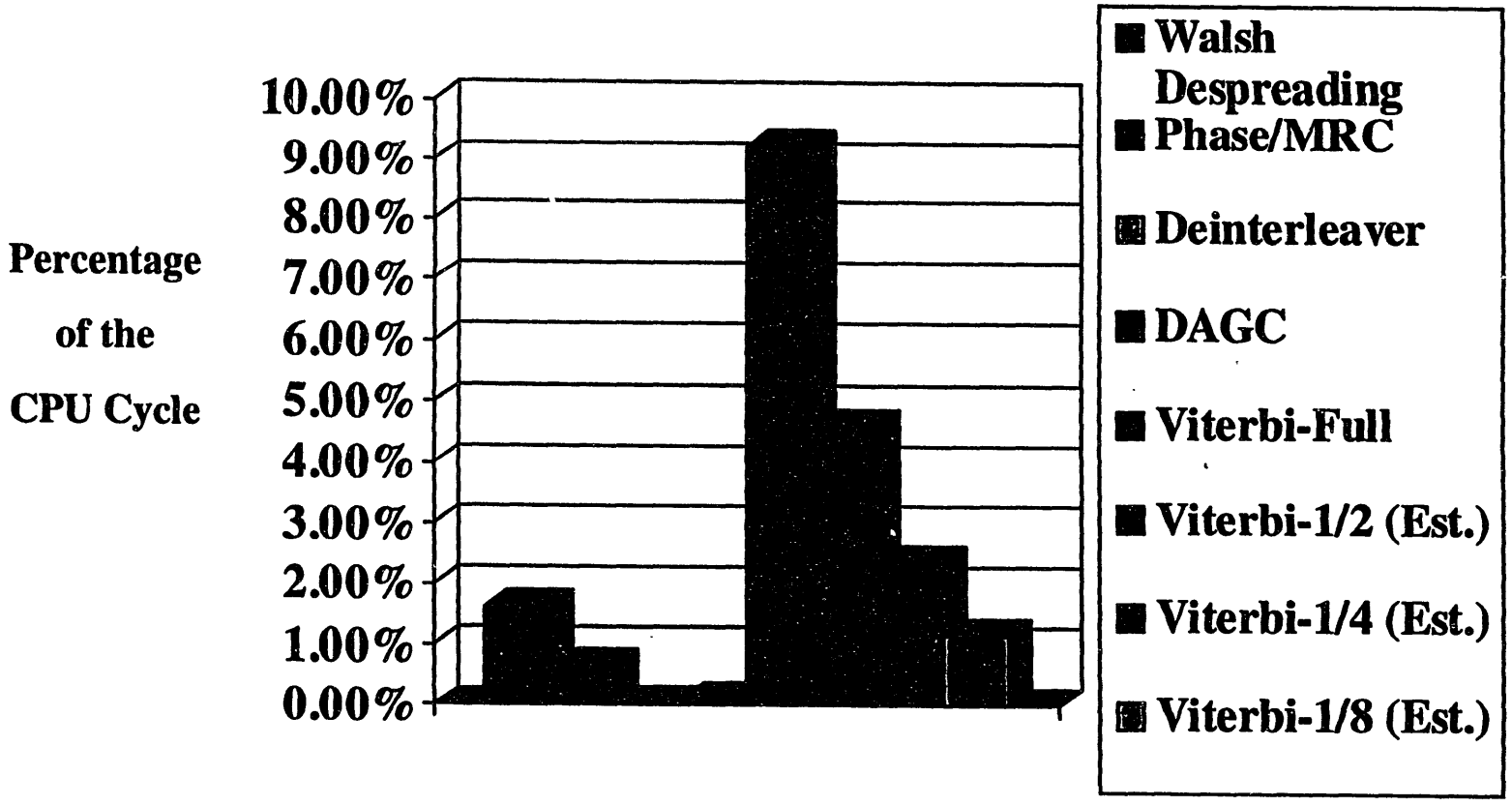


Figure 6.1. Process time share chart

In addition, the IS-95 system is a very complicated system and extremely computationally demanding. The transmission rate for an IS-95 system is 1.2288 Mcps. A series of digital processing need to be done on all those data. That is a significant amount of computation. Thus, the successful implementation of IS-95 on C6201 also proves that using software as an alternative implementation is not just possible for simple applications, it is also feasible for complicated systems such as IS-95, the industry standard of CDMA.

With respect to each of the research objectives, further conclusions may be drawn as follows:

6.1 Implementation of Walsh Despreading

Walsh Despreading has been successfully implemented on TI C6201 DSPs with an input chip rate of 1.2288 Mcps. The benchmark result for processing one frame of input signal (24,576 chips) is about 63,600 CPU cycles; this is approximately 1.59% of the CPU time.

Considering the very high input chip rate (1.2288 Mcps) and the huge 64x64 Walsh code matrix, the benchmark result of Walsh Despreading is very impressive and satisfactory. In order to achieve these successful results, efforts have been devoted to simplify the algorithms first, and look for regularity in the problem. By taking the advantage of the regularity of the Walsh code, surprisingly, even no memory storage is required for storing the code sequences in advance. This is a substantial saving for the very valuable on-chip memory compared to taking a 64x64 Walsh code matrix and storing it on the chip memory. Besides, appropriate code sequence is generated real time

based on the code sequence number requested by the user, which assumedly is already determined by the Rake Receiver.

6.2 Implementation of Phase Correction/MRC

The Phase Correction and Maximum Ratio Combining functions have been implemented in C on TI C6201 DSPs with great satisfaction. The benchmark result for its implementation is about 25,400 CPU cycles over one frame of data. This is only 0.64% of the processor time.

This implementation creates two circular buffers of size 16 each for the I and Q components of the pilot signal. It also creates two circular buffers of size 7 that are used to store the delayed I and Q symbols of the information signal. The Phase Correction & MRC algorithm involves the processing of the data frame for the I and Q branches of both pilot and information signals. There are 384 symbols for each type of signal per frame of data. However, by simplifying the algorithm, the efficiency has been significantly improved.

6.3 Implementation of Deinterleaver

The Deinterleaver function has been satisfactorily implemented in C on TI C6201 DSPs. The benchmark result for the Deinterleaver function to process one frame of data (384 symbols) is 1,440 CPU cycles; that is only 0.036% of the processor time, an extremely short time indeed.

The Deinterleaver has to generate a sequence array that rearranges the input sequence, which eventually could have a big code size. However, the detailed analysis of the data pattern demonstrates that the full rate Deinterleaver array, which seems to be in a

mess from appearance, has very unique regularity. The discoveries of the Rules "64", "32-16-48" and "1-3-2-4" have made this possible. Thus, the Interleaver array has been generated very efficiently with an especially easy form. This particular implementation does not even require any prior memory storage for the array and has a very small code size. Of course, the simpler the code, the faster it runs. All these attributes together make this implementation scheme great for DSP applications. As noticed during the programming process, without using these rules, the efficiency and performance would be substantially lower than what has been achieved now.

6.4 Implementation of DAGC

The Digital Automatic Gain Control function has been successfully implemented in C on TI C6201 DSPs. The benchmark result for the DAGC function to process one frame of data (384 symbols) is 4,116 CPU cycles, which is just 0.10% of the processor time. This is a very impressive and satisfactory result for the DAGC implementation.

Various specific techniques have been developed for this implementation. First, simplify the algorithm before programming. As shown in Figures 3.1 and 3.2, based on the regular implementation technique, the signal processing is very complicated, and requires substantial amount of floating point calculations over a sequence of 384 input symbols. No doubt, the resulted program from the DAGC Algorithms would be very time consuming, and greatly increase the error range of the results. However, after a series of effective conversation to simplify the DAGC Algorithms, the solutions turn out to be much simpler than they seem to be, as illustrated in Figures 3.3 and 3.4. Thus, it greatly simplifies the program and improves the efficiency. Combining with the other

novel approaches developed for this implementation such as using the C intrinsic functions to limit the floating calculations, significant improvements have been achieved.

The application of C intrinsic functions successfully change a rough floating point movement into a very simple fixed-point calculation. Thus, the most difficult challenge, handling a huge amount of floating point calculations, is successfully solved.

This series of conversions not only make the DAGC unit readily implemented on the C6201 DSP, and substantially minimize the floating point calculations, but also greatly increase the processing speed and reduce the error range.

6.5 Implementation of Viterbi Decoder

The Viterbi Decoder has been implemented on TI 6201 DSPs with impressive success. The benchmark result for decoding one frame of data (384 symbols) is 368.6 K cycles. This is about 9.22% of the CPU time. The Add-Compare-Select operation takes 889 cycles per trellis stage including overhead with a 4-cycle kernel per butterfly. The benchmark result compares very well to TI-internal hand scheduled assembly performance for the same type of Viterbi Decoder.

A lot of specific techniques have been developed in this project that are aimed specially to take the advantage of the C6201's architecture and its C Compiler in order to greatly raise the efficiency and meet the real time constraint imposed by the IS-95 system. These techniques include but not limited to the followings:

- Simplify the algorithms first;
- Changing butterfly() function calls to the for-loops;
- Skills on good indexing of the arrays;
- Keeping the pipelining flowing;

- Avoid using the standard C library function calls;
- Avoid using % for indexing; and
- Keep program simple

These discoveries and techniques bring a great simplification to the problem, substantially improve the program efficiency, and meet the real time constraint of the implementation of the Viterbi Decoder.

6.6 Implementation of PN

The Pseudorandom Noise (PN) Descrambling has been implemented on TI 6201 DSPs successfully. The PN generator in its current form takes approximately 1.2 million CPU cycles to generate one frame of the I and Q PN sequences (24,576 chips for each sequence); that is about 30% of the CPU time. It takes approximately another 51,000 CPU cycles for the demodulation, which is nearly 1.28% of the CPU time.

The bottleneck of the PN Descrambling function lies in generating the spreading quadrature sequences. Although it can still meet the real time constraint with this inefficiency, better alternatives should be investigated for the future work. The choice could be either a simplification of the algorithm or the implementation of the sequence generation function in a special hardware.

In summary, the IS-95 system, the industry standard of CDMA, has been successfully implemented entirely in C on TI fixed-point DSP TMS320C6201. Intensive efforts have been devoted to simplify the algorithms of concern based on their regularity and/or unique structure, and to develop many specific techniques in an effort to improve the efficiency of the program. The implementation has satisfied the real time constraint,

and achieved impressive efficiency. The benchmark results compare very well to TI-internal hand scheduled assembly performance of the same type of decoders. . IS-95 system is a very complicated system and extremely computationally demanding. This successful implementation of IS-95 provides a strong evidence for the possible use of general TI DSPs as an alternative to ASIC designs in the future, which would potentially benefit both customers and TI itself.

ABBREVATIONS AND SYMBOLS

ACS	Add-Compare-Select Operation
ASIC	Application Specific Integrated Circuit
AWGN	Additive White Gaussian Noise
CDMA	Code Division Multiple Access
CELP	Code Excited Linear Prediction
DAGC	Digital Automatic Gain Control
DSP	Digital Signal Processor
FDMA	Frequency Division Multiple Access
FEC	Forward Error-correcting (or Error Control) Coding
FTC	Forward Traffic Channel
GSM	Global System for Mobile
kbps	Kilo bits per second
ksps	Kilo symbols per second
mcps	1 million cycles per second
ML	Maximum Likelihood
msec	1 millionth second
MRC	Maximal Ratio Combining
PCS	Personal Communication System
PN	Pseudorandom Noise
SNR	Signal to Noise Ratio

TDMA	Time Division Multiple Access
TI	Texas Instruments
VA	Viterbi Algorithm
VLIW	Very Long Instruction Word

REFERENCES

1. Hendrix, H., *"Viterbi Decoding Techniques in the TMS320C54x Family,"* TI Technical Report, June 1996.

2. Papasakellariou, A., *"IS-95B Algorithm Description Document – DRAFT,"* TI Strictly Private, June 1998.

3. Proakis, J. G., *"Digital Communications,"* Third Edition, McGraw-Hill, Inc., New York, NY, 1995.

4. QUALCOMM Incorporated, *"CDMA Concepts and Terminology,"* Student Guide, R. M. Husseini, CDMA Training Department, December 1996

5. QUALCOMM Incorporated, *"IS-95A: The CDMA Standard,"* Student Guide, CDMA Technology Group, CDMA Training Department, 1997

6. Schildt, H., *"C: The Complete Reference,"* Third Edition, McGraw-Hill, Inc., New York, NY, 1995.

7. Shannon, C. E., *"Mathematical Theory of Communication,"* The University of Illinois Press, Urbana, Illinois, 1949.

8. Sklar, B., *"DIGITAL COMMUNICATIONS: Fundamentals and Applications,"* P T R Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

9. Texas Instruments, *"TMS320C62x/C67x CPU and Instruction Set,"* TI Product Reference Guide, 1998.

10. Viterbi, A. J., *"CDMA Principles of Spread Spectrum Communication,"* Addison-Wesley, Reading, Massachusetts, 1995.

APPENDICES

Appendix A: C Program Code for Walsh Despreading

```
/* Xiaozhen Zhang */
/* Jan 19, 1999 */
/* This code does Walsh despreading for a frame of IS95 Rate 1/2, K=9 convoluationally
encoded data. */
/* This version works! */
/* The benchmark for Walsh despreading over one frame of data is 63,612 CPU cycles */

#include <stdio.h>

#define WALSH_SIZE 64

typedef char Word;

static Word walsh[WALSH_SIZE];
static short win[WALSH_SIZE];
static short wout[384];
static Word num, count=0, class=0;

FILE *fp1;
FILE *fp2;

void generator(Word num);
void walsh_init(void);

void main(void)
{ int i, j, x;

 /* open output file for writing */
 if((fp1 = fopen("input.dat", "r"))==NULL) {
        printf("Cannot open input file.\n");
        exit(1);
        }

 /* open output file for writing */
 if((fp2 = fopen("output.dat", "w"))==NULL) {
        printf("Cannot open output file.\n");
        exit(1);
        }

 generator(8);  /* func generator() takes the desired Walsh code number as its parameter
*/

 for(x=0;x<2;x++){
```

```
        walsh_init();

        for (j=0; j<384; j++)
        { /* Read input chips */
         for (i=0; i<WALSH_SIZE; i++)
         { fscanf(fp1, "%d", &win[i]);
                }


       /* Multiply input chips with appropriate Walsh chips */
       for (i=0; i<WALSH_SIZE; i++)
          { wout[j] += (win[i])*(walsh[i]);
            }

          for (i=0; i<WALSH_SIZE; i++)
          { fscanf(fp1, "%d", &win[i]);
                }
       }

        for (j=0; j<384; j++)
     { fprintf(fp2, "%d\n", wout[j]);
      }

  if (feof(fp1)) {
    printf("End of file!\n");
    break;
    }
   }

 fclose(fp1);
 fclose(fp2);
 exit(0);
}

void generator(Word num)
{  int i = 0;
       if (num<0 || num >63)
              { printf("Please enter valid walsh code number!\n");
               exit(1);
                     }

       walsh[0] = 1;
       class = num%2;
       if (class == 0) walsh[1] = walsh[0];
        else walsh[1] = -walsh[0];
```

```
        class = num%4;
        for(i=2; i<4; i++)
                { if (class <= 1) walsh[i] = walsh[i-2];
                        else walsh[i] = -walsh[i-2];
                        }

        class = num%8;
        for(i=4; i<8; i++)
                { if (class <= 3) walsh[i] = walsh[i-4];
                  else walsh[i] = -walsh[i-4];
                  }

        class = num%16;
        for(i=8; i<16; i++)
                { if (class <= 7) walsh[i] = walsh[i-8];
                        else walsh[i] = -walsh[i-8];
                        }

        class = num%32;
        for(i=16; i<32; i++)
                { if (class <= 15) walsh[i] = walsh[i-16];
                        else walsh[i] = -walsh[i-16];
                        }

        for(i=32; i<64; i++)
                { if (num <= 31) walsh[i] = walsh[i-32];
                        else walsh[i] = -walsh[i-32];
                        }
}

void walsh_init(void)
{ int k;
 for (k=0; k<384; k++)
    { wout[k] = 0;
     }
 }
```

Appendix B: C Program Code for Phase Correction and MRC

```
/* Xiaozhen Zhang     August 20, 1998 */
/* This function does Phase Correction and Maximal Ratio Combining */
/* This version works! */
/* The benchmark is 25,413 CPU cycles for processing one frame of data */
/* The compiler flag used is -g -o3 -k -mg */

#include <stdio.h>

#define SIZE 16  /* MRC is done for summing over 16 pilot symbols */
#define DELAY 7  /* Delay is 7 symbols */
#define FULL_FRAME 384

static int pbuffer_I[SIZE], pbuffer_Q[SIZE], sbuffer_I[DELAY], sbuffer_Q[DELAY];
static int avg_I, avg_Q, I_term, Q_term;
static int mrc;

void init_pilot(void);

FILE *fps_i, *fps_q, *fpp_i, *fpp_q, *fp3;

void main(void)
{int j=0, i=0, count=0;

        /* open input file for reading */
        if ((fps_i = fopen("mrc_in_i.dat", "r"))==NULL) {
           printf("Cannot open real input signal file.\n");
           exit(1);
          }

        if ((fps_q = fopen("mrc_in_q.dat", "r"))==NULL) {
           printf("Cannot open imaginary pilot signal file.\n");
           exit(1);
          }

        /* open real part of the pilot signal file for reading */
        if ((fpp_i = fopen("i_pilot.dat", "r"))==NULL) {
           printf("Cannot open real pilot signal file.\n");
           exit(1);
          }

        /* open imaginary part of the pilot signal file for reading */
        if ((fpp_q = fopen("q_pilot.dat", "r"))==NULL) {
           printf("Cannot open imaginary pilot signal file.\n");
```

```
        exit(1);
       }

      /* open output file for writing */
      if ((fp3 = fopen("mrc_output.dat", "w"))==NULL) {
        printf("Cannot open output file.\n");
        exit(1);
       }

      init_pilot();

      for(;;){
            avg_I = avg_I - pbuffer_I[i]; /* pilot signals are averaged over 16 symbols
*/
            fscanf(fpp_i, "%d", &pbuffer_I[i]); /* reading on-time symbol for the
pilot's real term */
            if(feof(fpp_i))
      { if (count == 0)
       { printf("End of file - Frame completed!\n");
        } else printf("End of file - Frame incompleted!\n");
     break;
     }
            avg_I = avg_I + pbuffer_I[i]; /* pbuffer_I contains the real term of the
pilot signals */

            I_term = sbuffer_I[j]*avg_I; /*sbuffer_I contains the real term of the
received singnals */
            fscanf(fps_i, "%d", &sbuffer_I[j]); /* reading delayed symbol (by number
of DELAY) for the

      received signal's real term */

            avg_Q = avg_Q - pbuffer_Q[i];
            fscanf(fpp_q, "%d", &pbuffer_Q[i]); /* reading on-time symbol for the
pilot's imaginary term */
            avg_Q = avg_Q + pbuffer_Q[i];  /* pbuffer_Q contains the imaginary term
of the pilot signals */

            Q_term = sbuffer_Q[j]*avg_Q;    /*sbuffer_I contains the imaginary term
of the received singnals */
            fscanf(fps_q, "%d", &sbuffer_Q[j]); /* reading delayed symbol (by
number of DELAY) for the

      received signal's imaginary term */

            mrc = I_term + Q_term;    /* the maximal ratio combining output */
```

```
                fprintf(fp3, "%d\n", mrc);  /* writing the MRC results to an output file */

      count = (count+1)%FULL_FRAME; /* updating the Frame Counter */
                i = (i+1)%SIZE;  /* updating the circular buffer for the received signals */
                j = (j+1)%DELAY;   /* updating the circular buffer for the pilot signals */
      }
    fclose(fps_i);
    fclose(fps_q);
    fclose(fpp_i);
    fclose(fpp_q);
    fclose(fp3);
    exit(0);
  }

void init_pilot(void)
{int r;
 int m;
 for (r=0; r<=SIZE-1; r++)   /* Initialization for the buffer -- IMPORTANT!! */
   {pbuffer_I[r]=0;          /* For the first 16 symbols, MRC is done */
    pbuffer_Q[r]=0;          /* for summing over less than 17 symbols */
    }
 for (m=0; m<=DELAY-1; m++)
   {sbuffer_I[m]=0;          /* Initilizing the buffers to 0 for the effect of delay by DELAY.
*/
    sbuffer_Q[m]=0;
    }
 avg_I=0;       /* Initalizing the sum of pilot symbols -- IMPORTANT!! */
 avg_Q=0;
 }
```

Appendix C: C Program Code for Deinterleaver

```
/* Background:  IS95 rate 1/2, K=9 convolutional encoder */
/* This function does deinterleaving over one frame of IS95 data */

/* Xiaozhen Zhang */
/* January 20, 1999 */

/* Benchmark = 1,440 CPU cycles using -gk -mg -o3 as the compiler flag*/

#include <stdio.h>
#define FRAME 384

int order[FRAME];
FILE *fp1;    /* fp1 is the file pointer for the input file; */
FILE *fp2;    /* fp2 is the file pointer for the output file; */

void init(void);

void main(void)
{ int in[FRAME], out[FRAME];
  int i, k;

        /* open input file for reading */
        if ((fp1 = fopen("input.dat", "r"))==NULL) {
           printf("Cannot open input file.\n");
           exit(1);
          }

        /* open output file for writing */
        if ((fp2 = fopen("output.dat", "w"))==NULL) {
           printf("Cannot open output file.\n");
           exit(1);
          }

        for(i=0;i<FRAME;i++){
                /* Read input symbols */
                fscanf(fp1, "%d", &in[i]);
     }

   init();

/* This for loop changes the sequence of the input symbols */
/*  and put them into the correct position */
   for (k=0;k<FRAME;k++) {
```

```
        out[(order[k])]=in[k];
        }

    for (i=0; i<FRAME;i++) {
            fprintf(fp2, "%d\n", out[i]);
                }

    fclose(fp1);
    fclose(fp2);
    printf("End of file - Frame completed!\n");
    exit(0);
 }

/* function init() generates the Full Rate Interleaver Output Array */

void init(void)
{int i, j;

/* order[] yields the sequence order for the output array */
/* Locate the position for 0 through 3 in the array*/
        for (i=0; i<=1; i++) {
                order[i*192] = i;
                order[i*192+96] = i+2;
      }

/* Locate the position for 4 through 15 in the array */
    for (i=0; i<=3; i++) {
       order[i*96+24] = order[i*96]+8;
       order[i*96+48] = order[i*96]+4;
       order[i*96+72] = order[i*96]+12;
       }
/* Locate the position for the remaining part of the 384 symbols */
    for (i=0; i<=15; i++) {
       order[i*24+6] = order[i*24]+32;
       order[i*24+12] = order[i*24]+16;
       order[i*24+18] = order[i*24]+48;
       }

    for (j=0; j<=63; j++) {
       for (i=1; i<=5; i++)
          order[j*6+i] = order[j*6]+i*64;
       }
 }
```

Appendix D: C Program Code for DAGC

```
/* Xiaozhen Zhang */
/* December 23, 1998 */
/* Digital Automatic Gain Control (DAGC)  */
/* This version works!  It yields a cycle count = 4116 cpu cycles*/

#include <stdio.h>
#define FULL 384

int dagc_in[FULL], mag[FULL];
short int dagc_out[FULL];
int i, j, coeff;
unsigned int sum;
FILE *fp1, *fp2;

void main(void)
{
        if ((fp1 = fopen("input.dat", "r"))==NULL) {
                printf("Cannot open input file.\n");
                exit(1);
          }

        /* open output file for writing */
        if ((fp2 = fopen("output.dat", "w"))==NULL) {
           printf("Cannot open output file.\n");
           exit(1);
           }

        for(;;){
                for (j=0; j<FULL; j++)
                {
                        /* Read input symbol pair */
                        fscanf(fp1, "%d", &dagc_in[j]);
                        if(feof(fp1)) break;
                        }

                if(feof(fp1))
          { printf("End of file\n");
        break;
                }

                sum = 0;
                for (i=0; i<FULL; i++) {
                mag[i] = _abs(dagc_in[i]);  /* C6x C Compiler intrinsic _abs */
```

```
            /* _abs returns the saturated absolute value of the source */
                sum += 31 - _lmbd(1, mag[i]); /* C6x intrinsic _lmbd */
                }       /* _lmbd searches for a leftmost 1 & returns the # of bits up to the
bit change. */

                coeff = (int)(sum/384);

                for (i=0; i<FULL; i++) {
                   dagc_out[i] = dagc_in[i] >> coeff;
                   }

        for (i=0; i<FULL; i++) {
        if (dagc_in[i]<0) dagc_out[i] += 1;
                        }

                for(j=0; j<FULL; j++) {
             fprintf(fp2, "%d\n", dagc_out[j]);
             }
          }

        fclose(fp1);
        fclose(fp2);
        exit(0);
 }
```

Appendix E: C Program Code for Viterbi

```
/* Xiaozhen Zhang  December 23, 1998 */

/* Viterbi Decoder for K=9 and rate=1/2 convolutional code*/
/* The Viterbi Decoder is designed for decoding the data frame at full rate */
/* The code polynomails for the IS-95-A forward link code
  are octal 561 and 753 */

/* The benchmark result is 368,625 CPU cycles for decoding one frame of data (384
symbols) */
/* The compiler flag used is -g -o3 -k -mg */
/* There is a related documentation file viterbi.doc written by Xiaozhen Zhang*/
/* viterbi.doc is written to specifically address the design choices made during the
implementation */

#include <stdio.h>

#define NODES 256
#define MEMPATH 50
#define MERGEDIST 48  /* trace back length is the smallest even number */
                      /*  greater than 5 times the constraint length */
#define FRAME 192

int counter = 0, wd = 0;
int sd[384], data[2], data2[MEMPATH];
short int dm[128], cmetric[NODES], nmetric[NODES];
unsigned int paths[8*MEMPATH];  /* memory storage for the decoded bit at each
                                                    stage is 256 bits = 8*32bits = 8
words */
void init(void);
void branchmetric(int w);
static void traceback1(int world);
static void traceback2(int world);

FILE *fp1;    /* fp1 is the file pointer for the input file; */
FILE *fp2;    /* fp2 is the file pointer for the output file; */


void main(void)
{ int a, k, p;
 short int dmb;

        init();
```

```
        /* open input file for reading */
        if ((fp1 = fopen("input.dat", "r"))==NULL) {
           printf("Cannot open input file.\n");
           exit(1);
          }

        /* open output file for writing */
        if ((fp2 = fopen("output.dat", "w"))==NULL) {
           printf("Cannot open output file.\n");
           exit(1);
           }

        for(;;){

        for (a=0; a<384; a++)
           {
                /* Read input symbol pair */
                fscanf(fp1, "%d", &sd[a]);
                if(feof(fp1)) break;
                }

        if(feof(fp1))
    { if (counter == 0) printf("End of file - Frame completed!\n");
                else printf("End of file - Frame incompleted!\n");
     break;
     }

   for (p=0;p<=95;p++) {
/* Compute branch metrics using the input symbol pair */
        branchmetric(counter);

/* Metric Update */
/* The following loop does the add-compare-select(ACS) operation, it takes the old
metrics from cmetric and stores the new metrics into nmetric. */
        for (a = 0; a < 8; a++)
           { int m00, m01, m10, m11;
            unsigned int dec0 = 0;
            unsigned int dec1 = 0;

            int path_bit = 1;

            for (k = 0; k < 16; k++)
               {dmb = dm[16*a+k];
                   m00 = cmetric[16*a+k] + dmb;
                   m01 = cmetric[16*a+k+128] - dmb;
                   m10 = cmetric[16*a+k] - dmb;
```

```
                m11 = cmetric[16*a+k+128] + dmb;

                if (m01>m00) dec0 += path_bit;
                  else m01 = m00;
                if (m11>m10) dec1 += path_bit;
                  else m11 = m10;

                nmetric[2*(16*a+k)] = (short)m01;
                nmetric[2*(16*a+k)+1] = (short)m11;

                path_bit *=4;
                }
              paths[8*wd+a] = dec0|(dec1*2);
          }

        wd++;
        counter++;

        /* Compute branch metric */
        branchmetric(counter);

/* Metric Update */
/* The following loop also does the add-compare-select(ACS) operation, except it takes
the old
metrics from nmetric and stores the new metrics into cmetric. */
        for (a = 0; a < 8; a++)
           { int m00, m01, m10, m11;
             unsigned int dec0 = 0;
             unsigned int dec1 = 0;
             int path_bit = 1;

             for (k = 0; k < 16; k++)
                {dmb = dm[16*a+k];
                   m00 = nmetric[16*a+k] + dmb;
                   m01 = nmetric[16*a+k+128] - dmb;
                   m10 = nmetric[16*a+k] - dmb;
                   m11 = nmetric[16*a+k+128] + dmb;

                   if (m01>m00) dec0 += path_bit;
                     else m01 = m00;
                   if (m11>m10) dec1 += path_bit;
                     else m11 = m10;

                   cmetric[2*(16*a+k)] = (short)m01;
                   cmetric[2*(16*a+k)+1] = (short)m11;
```

```
                path_bit *=4;
                }
            paths[8*wd+a] = dec0|(dec1*2);
            }

        if (++wd >= MEMPATH) wd -= MEMPATH;   /* Implementing circular buffer
storage */
        counter++;
        if(counter==FRAME) traceback2(wd);
        if(counter>=MEMPATH)
         traceback1(wd);
         }
        }
  fclose(fp1);
  fclose(fp2);
  exit(0);
 }

/* Initializing the matrices. */
void init(void)
 { int k;

        for(k=0;k<NODES;k++){
        cmetric[k] = 0;
        nmetric[k] = 0;}

        for(k=0;k<8*MEMPATH;k++){
        paths[k] = 0;}

        data[0] = 0;
        data[1] = 0;
        }

void branchmetric(int w)
 { int y, k;
  short int d0, d1, d2, d3;
        /* Compute branch metric */
  d0 = (short)(sd[2*w] + sd[2*w+1]);      /* M */
  d1 = (short)(sd[2*w] - sd[2*w+1]);      /* L */
  d2 = - d1;                                                    /* -L */
  d3 = - d0;                                                /* -M */

        for (k=0; k<33; k=k+32) {
                for (y=0; y<7; y=y+6)
                        { dm[y+k] = d0;       /* 0, 6, 32, 38 */
                         dm[y+k+1] = d2;      /* 1, 7, 33, 39 */
```

```
                    dm[y+k+8] = d1;      /* 8, 14, 40, 46 */
                    dm[y+k+9] = d3;      /* 9, 15, 41, 47 */
                    }

            for (y=0; y<3; y=y+2)
                    { dm[y+k+2] = d3;      /* 2, 4, 34, 36 */
                     dm[y+k+3] = d1;      /* 3, 5, 35, 37 */
                     dm[y+k+10] = d2;     /* 10, 12, 42, 44 */
                     dm[y+k+11] = d0;     /* 11, 13, 43, 45 */
                     }

            for (y=0; y<7; y=y+6)
               { dm[y+k+16] = d2;     /* 16, 22, 48, 54 */
                     dm[y+k+17] = d0;     /* 17, 23, 49, 55 */
                     dm[y+k+24] = d3;     /* 24, 30, 56, 62 */
                     dm[y+k+25] = d1;     /* 25, 31, 57, 63 */
                     }

            for (y=0; y<3; y=y+2)
                    { dm[y+k+18] = d1;          /* 18, 20, 50, 52 */
                     dm[y+k+19] = d3;     /* 19, 21, 51, 53 */
                     dm[y+k+26] = d0;     /* 26, 28, 58, 60 */
                     dm[y+k+27] = d2;     /* 27, 29, 59, 61 */
                     }
          }

        for (y=0; y<32; y++)
        { dm[y+64] = dm[y+16];
         dm[y+96] = dm[y+16];
         }
 }

/* Traceback */
static void traceback1(int world)
{       int ma, i, j, best_state, bit, test, m;
   short int mx, *c;
        /* Find the best state for the current state */
        ma = 0;
        mx = cmetric[0];

        for (m=1; m<=256; m++)
           if (mx < cmetric[m] ) mx = cmetric[ma=m];

        c = &cmetric[ma];
        best_state = c - cmetric;
```

```
        /* Trace back for the best state for MERGEDIST number of times */
        if (--world<0) world += MEMPATH;

        for(i=0; i<MERGEDIST; i++) {
            test= paths[8*world + (best_state>>5)] & (1<<(best_state & 31)) ;
                if (test) best_state = best_state + 256;
                best_state = (best_state >> 1);
                if (--world<0) world += MEMPATH;
               }

   /* Best state is now the encoder state MERGEDIST bits back.
     Continue to trace back until we accumulate 2 bits. */

        for(j=0; j<=1; j++) {
          if (paths[8*world + (best_state>>5)] & (1<<(best_state & 31))) {
              best_state = best_state + 256;
              data[j] = 1; } else data[j]=0;
          best_state = (best_state >> 1);
                if (--world<0) world += MEMPATH;
          }
          fprintf(fp2, "%d\n", data[1]);
          fprintf(fp2, "%d\n", data[0]);
 }

static void traceback2(int world)
{  int j, best_state;
        /* Find the best state for the current state */
        /* Trace back for the best state for MERGEDIST number of times */
        counter = 0;
        best_state = 0;
        if (--world<0) world += MEMPATH;

        for(j=0; j<MEMPATH; j++) {
          if (paths[8*world + (best_state>>5)] & (1<<(best_state & 31))) {
              best_state = best_state + 256;
              data2[j] = 1; } else data2[j]=0;
          best_state = (best_state >> 1);
          if (--world<0) world += MEMPATH;
          }

        for(j=0; j<MEMPATH; j++) {
          fprintf(fp2, "%d\n", data2[MEMPATH-1-j]);
          }
        wd = 0;
        init();
 }
```

Appendix F: C Program Code for PN

```
/****PN despreading *********/

/*************************************/
/*  Xiaozhen Zhang                   */
/*  July 15, 1998                    */
/*  C Source File for PN Despreading */
/*  Part I of Thesis Project:        */
/*    Implementing IS95 on C6x       */
/*************************************/

/* This version works! */
/* Jan. 19, 1999 */


#include "pn_defs.h"

UWord i_pn_reg;
UWord q_pn_reg;
UWord i_gen_poly;
UWord q_gen_poly;

void init_pn(UWord I_Init_State, UWord Q_Init_State)
{
 /*********************************************
  * Initialize the I and Q LFSR's.
  *********************************************/

  i_pn_reg = I_Init_State;
  q_pn_reg = Q_Init_State;

 /*********************************************
  * Initialize the generator polynomials with the
  * the 15th order primitive polynomials with a
  * shift (because they are applied to bits 14:1).
  *********************************************/
  i_gen_poly = PRIMITIVE_POLY_1 << 1;

  q_gen_poly = PRIMITIVE_POLY_2 << 1;
}

/*****************************************************************
 * This function generates the I and Q PN sequences from the PN
 * linear feedback shift registers (LFSR's) of length 15. Bit
```

```
 * 0 of the data structure holds the MSB of the LFSR, since it
 * is shifted from left to right.
 *
 * The PN sequences have a period of 2^15 due to zero insertion,
 * and output at the chip rate of 1.2288 MHz.
 *
 * Params (input):
 *    pn_load  - If TRUE, a new state is loaded into IQ PN (with an
 *               effective delay of 1 chip). If FALSE, the next two
 *               input parameters are ignored and the PN LFSR's function
 *               normally.
 *    new_I_state - New state for loading into I PN sequence LFSR.
 *    new_Q_state - New state for loading into Q PN sequence LFSR.
 *
 * Params (output):
 *      pn_I_ptr -  the output of I PN generator.
 *      pn_Q_ptr -  the output of Q PN generator.
 ****************************************************************/

void pn_seq(UWord new_I_state, UWord new_Q_state,
        UWord *pn_I_ptr, UWord *pn_Q_ptr)
{
UWord reg_msb;
UWord nor_res;

/*************************************************************
 * First calculate the output of the I register using the MSB,
 * XOR'ed with the NOR result of the lower 14 bits of the LFSR.
 * This is done to implement zero insertion after state 2^15-1.
 *************************************************************/

 reg_msb = i_pn_reg & LSB_MASK;

 if ((i_pn_reg >> 1) == 0)
  nor_res = 1;

 else
  nor_res = 0;

 *pn_I_ptr = reg_msb ^ nor_res;

/*************************************************************
 * Repeat this process for the Q LFSR to get the PN output.
 *************************************************************/

 reg_msb = q_pn_reg & LSB_MASK;
```

```
  if ((q_pn_reg >> 1) == 0)
    nor_res = 1;

  else
    nor_res = 0;

  *pn_Q_ptr = reg_msb ^ nor_res;

 /*****************************************************************
  * Update the I and Q LFSR's using the proper polynomials.
  *****************************************************************/

  if (*pn_I_ptr)

    i_pn_reg = (i_pn_reg ^ i_gen_poly);

  if (*pn_Q_ptr)

    q_pn_reg = (q_pn_reg ^ q_gen_poly);

  /*****************************************************************
  * Implement feedback to update LSB of the LFSR.
  *****************************************************************/

    i_pn_reg = (*pn_I_ptr << (PN_REG_LEN-1)) | (i_pn_reg >> 1);

    q_pn_reg = (*pn_Q_ptr << (PN_REG_LEN-1)) | (q_pn_reg >> 1);
}

void main()
{
UWord pnI, pnQ;
UWord pn_count;
  init_pn(0, 0);

  for (pn_count=0; pn_count < NUM_PN_OUTPUTS; pn_count++)
  {
    pn_seq(0, 0, &pnI, &pnQ);
  }
}
```

THESIS PROCESSING SLIP

FIXED FIELD: ill. _____ name _____

index _____ biblio _____

► COPIES: (Archives) Aero Dewey (Eng) Hum

Lindgren Music Rotch Science

TITLE VARIES: ► [L] see degree book

NAME VARIES: ► [ ] _____

IMPRINT: (COPYRIGHT) _____

► COLLATION: 102 l

► ADD: DEGREE: S.B. ► DEPT.: E.E.

SUPERVISORS: _____

NOTES:

cat'r: date:

page: 540

► DEPT: E.E. ► J155

► YEAR: 1999 ► DEGREE: M.Eng.

► NAME: ZHANG, Xiaozhen