DSP SPECIFIC OPTIMIZED IMPLEMENTATION OF VITERBI

By

Yame Asfia

Submitted to the department of Computer Engineering in fulfillment of the requirements for the degree of

Masters in Science

in

Computer Engineering

External Supervisor

Dr. Muid-ur-Rahman Mufti

Internal Supervisor

Brig Dr. M. Younus Javed

College of Electrical and Mechanical Engg,

National University of Sciences and Technology

2009

ACKNOWLEDGEMENT

All praises and thanks to **ALMIGHTY ALLAH,** Who made this difficult work possible. This research work is the product of His blessings.

I would like to express my heartfelt gratitude with love to my family especially to my parents for being so supportive and caring throughout the course of this study, my brother and my sisters without whom I would have never be able to achieve this milestone.

I extend my appreciation and thanks to my supervisor, Dr. Muid-ur-Rahman Mufti, Professor, Department of Computer Engineering, UET Taxila for his long lasting guidance, positive criticism and kind cooperation offered to me during my MS tenure. I feel I may never be able to pay him the deserved thanks. I wish to express my heartiest thanks to my internal-supervisor, Brig Dr. M. Younus Javed, Head of the Department, College of Electrical and Mechanical Engg, NUST Rawalpindi, for his kind cooperation and constructive suggestions at every step of my research.

The constant advice and encouragement of my friends and the work of early scientists which blazed this study is also highly acknowledged.

Last but not the least, I appreciate all those who remember me in their prayers and encourage me throughout the deals of my life.

YAME ASFIA

… Dedicated to my parents

ABSTRACT

Due to the rapid changing and flexibility of Wireless Communication protocols, there is a desire to move from hardware to software/firmware implementation in DSPs. High data rate requirements suggest highly optimized firmware implementation in terms of execution speed and high memory requirements. This work suggests optimization levels for the implementation of a viable Viterbi decoding algorithm on a commercial off-the-shelf DSP. Viterbi is a popular decoding algorithm employed in building up many communication systems because of its robustness and ability to detect and correct most of the transmission errors. Optimizing this algorithm to achieve maximum execution speed and minimum possible memory usage is the focus of our research. Different Logic and Code optimization techniques have been applied and discussed throughout the document. With an aim of producing an improved design (in terms of execution speed and memory requirements), the proposed system is able to deliver a data rate of 1.7 Mbps on a clock frequency of 600MHz, for a constraint length (i.e. K=7).

Table of Contents

Table of Figures

Table of Tables

CHAPTER – 1 : INTRODUCTION

For the last few decades, there has been an immense research in the field of embedded system designing. With an ambition of achieving compaction and flexibility in designs, designers are focusing on the firmware implementation of algorithms on DSPs. The advancements in technology, call for high data rate systems and so improving the execution speed of such algorithms is critical and unavoidable. Moreover, the memory requirements of such systems need to be compromised in a positive manner. For the achievement of such goals, different kinds of optimization techniques need to be applied suggesting an optimal implementation. This work focuses on the implementation of a ½ code rate Viterbi decoder with a constraint length K=7 on ADSP-BF533. A comparison of this implementation with the designs implemented on TMS320C6201, having different constraint lengths, has also been provided.

1.1 Challenges vs. Technology

Any embedded application attempts to minimize, simultaneously, four factors:

- The number of transistors employed, which impacts die and package size, unit cost and power consumption. Advances in process technology continuously reduce transistor area, but both static and dynamic power consumption depend on the transistor count. The transistor count remains an important metric of system efficiency.
- The number of clock cycles required, which impacts performance and power consumption. Increasing clock frequencies associated with smaller process geometries permit more clock cycles in a given time interval, but at the expense of increased power consumption. Fewer clock cycles means less power consumption.
- The time taken to develop the application, which strongly influences its market acceptance. A product that misses its market window is a total waste of

development effort. In many cases software development takes more time and costs more than hardware development.

- Nonrecurring engineering (NRE) costs such as mask manufacturing and the cost of hardware and software development. The increased NRE costs associated with leading-edge process technologies are putting these out of reach for many applications.

1.1.1 Available Technology

Generally four kinds of hardware platforms are available for the implementation of any communication or signal processing system/algorithm. These include ASICs, MCUs, FPGAs and DSPs. Depending on the type of application and the system requirements i.e. performance and flexibility, developers have to make a sound choice among these platforms.

1. ***Application Specific Integrated Circuits (ASICs)***

   An ASIC is custom-designed for a particular application, possibly embedding one or more MCU or DSP cores, with as much as possible of the total system functionality implemented on a single die.

   This optimizes the number of transistors and clock cycles (and therefore unit cost and power consumption), at the expense of development time and NRE cost that are generally an order of magnitude higher than those for MCUs, DSPs or FPGAs.

   Application-specific functions, in particular analog operations, must often be implemented off-chip. Die size, package size, pin-out and power consumption are less than optimal compared with what can be achieved by this technology, namely ASICs.

2. ***Microcontrollers (MCUs)***

   Microcontrollers (MCUs) are general-purpose devices for information processing and control that can be adapted to a wide variety of applications by software. Application development effort is limited to software development and validation, and NRE costs are amortized amongst all the users of a particular MCU architecture. Clock cycle optimization is determined by code optimization, and the code footprint influences the number of transistors required for memories. Compact code that makes the most efficient use of the MCU architecture is essential. MCUs generally use transistors and clock cycles efficiently, but not optimally.

3. ***Field programmable gate arrays (FPGAs)***

   Field programmable gate arrays (FPGAs) limit development effort to the coding required to configure them, and share NRE costs amongst a very large population of users, at the expense of a high level of transistor redundancy (and therefore high unit costs) and a limited optimization of clock cycles. Power consumption is far from optimal.

4. ***Digital signal processors (DSPs)***

   Digital signal processors (DSPs) hard-wire the basic functions of many signal-processing algorithms. This optimizes transistor use and clock cycles for the required operations, at the expense of flexibility. Code is simpler than that required for MCUs. In many cases a DSP is an optimal solution for some but not all of the functions required of an application. Many MCUs include basic DSP operations in their instruction set, which enables them to do simple signal processing, without the need for a dedicated DSP.

1.1.2 Technology tradeoffs

The four technologies represent different tradeoffs towards achieving the four optimizations. The choice for any particular application is an engineering compromise. In most cases, the choice depends on a complex combination of factors, and no single technology is ideal. Different technology mixes are often most appropriate at different stages of the lifecycle of the end-user product. During prototyping and production ramp-up an FPGA or MCU/DSP-plus-FPGA solution may be preferable, in order to reduce development time and cost. When the product goes into high volume, its functionality can be re-mapped into an ASIC that embeds the MCU or DSP core from the standard product, and absorbs the logic from the FPGA, thereby optimizing die size, unit cost, clock cycles and power consumption without the need to rewrite the software. The high NRE costs associated with ASIC development are amortized over the high production volume.

Where, FPGAs and ASICs are considered superior in terms of performance and power efficiency, they greatly lack the programming flexibility, offered by DSPs, which is highly desirable in implementing complex signal processing algorithms. Moreover, the highly optimized architectural features and less price premium than FPGAs, make DSPs the right choice for their use in the signal processing and communication applications. A detailed comparison of all these platforms can be found in [4].

1.2 Viterbi Decoder

The heart of this discussion is the DSP specific implementation of Viterbi decoder, which is a well known communication algorithm. Viterbi is employed for decoding the convolutionally encoded data, which has been transmitted through an Additive White Gaussian Noise ***(AWGN)*** channel. The strength of this algorithm to correct most of the errors makes it an essential module of a CDMA/WCDMA system.

"Viterbi Algorithm (VA) decoders are currently used in about one billion cell phones, which is probably the largest number in any application. However, the largest current consumer of VA processor cycles is probably digital video broadcasting. A recent estimate at Qualcomm is that approximately 1015 bits per second are now being decoded by the VA in digital TV sets around the world, every second of every day" [2].

The algorithm works by forming a trellis structure which is eventually traced back, for decoding the received information. This calls for massive storage requirements. Furthermore, the emerging Wireless standards, which deliver high data rates, have also raised the performance bar for Viterbi and hence the need for an optimized system that improves the execution speed and uses memory optimally, by reducing the logic and code complexity, is eminent.

1.3 Related Work

[3] has presented an implementation on TMS320C6201 for a constraint length K=9 and code rate= 1/3. The design of a 500MHz, two 8-state, 7-bit soft output Viterbi decoders matched to an EPR4 channel and a rate-8/9 Convolutional code,

implemented in 0.18µm CMOS technology has been described in [5]. [6] Describes the design and implementation of a 19.2 kbps, 256 state Viterbi decoder using FPGAs with the added capability of catering to higher input data rates. To the best of our knowledge no research work has focused on the implementation of Viterbi on a general DSP platform. Here we discuss a generalized Viterbi implementation which has been optimized well, to suit any DSP platform (processor).

1.4 Thesis Organization

The discussion has been organized as follows:

Chapter II:

Chapter II explains the presents the discussion for Convolutional encoding followed by Viterbi decoding. This chapter briefly provides a background of the modules of a typical communication model employing Convolutional encoding and Viterbi decoding. The channel modes are also discussed.

Chapter III:

In this chapter we brief the architecture of **ADSP-BF533 Blackfin®** Processor.

Chapter IV:

Chapter IV explains our optimization strategy. The optimization techniques in practice and those devised by us are thoroughly discussed.

Chapter V:

The last chapter carries out the implementation details of the system model. It thoroughly explains the system design and its components. Different phases of the design process are also described.

Chapter VI:

The last chapter provides the simulation results and rates the performance of the optimized design.

CHAPTER – 2 : CONVOLUTIONAL ENCODING AND VITERBI DECODING

Forward Error Correction schemes are largely employed to detect and correct errors in the data transmitted through any communication channel. Such schemes introduce some redundant bits in the original stream of data. This redundancy helps not only in detecting the errors but also proves to be helpful in locating and correcting the error within some limited precision. This chapter briefs out the details of the Forward Error Correction encoding scheme ***Convolutional Encoding*** and the decoding algorithm used on the receiving end called ***Viterbi Decoding***.

2.1 Convolutional Encoding

Convolutional coding was introduced in 1955 as a strong FEC (Forward Error Correction) coding scheme. This FEC scheme treats the incoming data in streams rather than in blocks.

2.1.1 Structure

The Encoding process involves a shift register accompanied by modulo-2 addition logic. The length of shift registers is called the constraint length ***'K'*** of the decoder so that the system has $\mathbf{2^{(K-1)}}$ states. The number of different modulo-2 addition combinatorial logics determine the code rate **'k/n'** i.e. ***ratio of number of input bits to the number of output bits*** of the decoder [1].

Figure-2-1 shows the block diagram of a feed forward convolutional encoder with a constraint length **K=3** and a code rate of **n=1/2**. It means that the system has a total of $2^{(K-1)} = 2^{(3-1)} = 4$ states and for every single input bit, the system produces two output bits. The inputs to the two adders (XOR) form the generator polynomials [7,5]. As the shift registers are initially cleared, the initial state is **'0'**. Once a pair of output bits has been produced, the data is shifted right to introduce the next input bit and then the same encoding process is applied over this bit as well. Hence the encoded output is a function of the current input and ***K-1 (3-1= 2)*** previous inputs. The process may be terminated once all the input bits have been processed or some extra zero bits may be added so that the encoding process ends in all zero state. Our discussion will not be covering the details of these termination decisions.

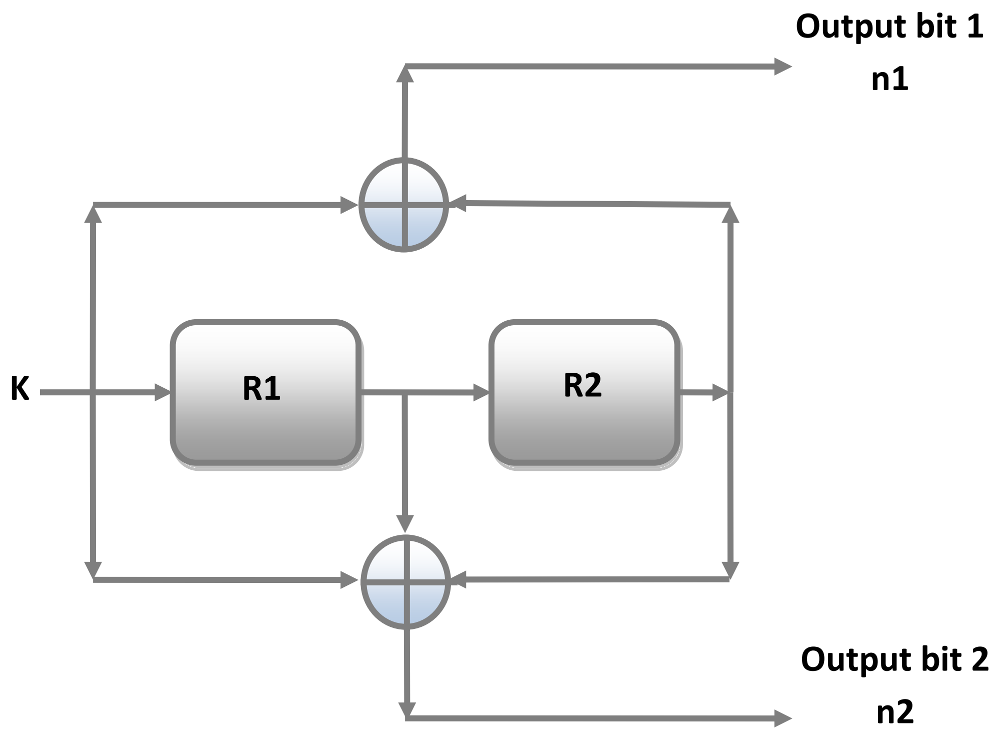


Figure 2-1: Block Diagram of Convolutional Encoder [1]

2.1.2 State Diagram

The whole encoding process may be best explained with the help of this state diagram (Figure-2-2). The state diagram shows all the possible transitions and outputs for each state-input pair.

As already explained, the initial state is **'00'**. The edges represent the state transitions where the dotted ones indicate the case for input bit value equal to 1 and the solid lines represent the transitions for 0 bit. The corresponding two bit outputs are labeled on the relevant edges [1].

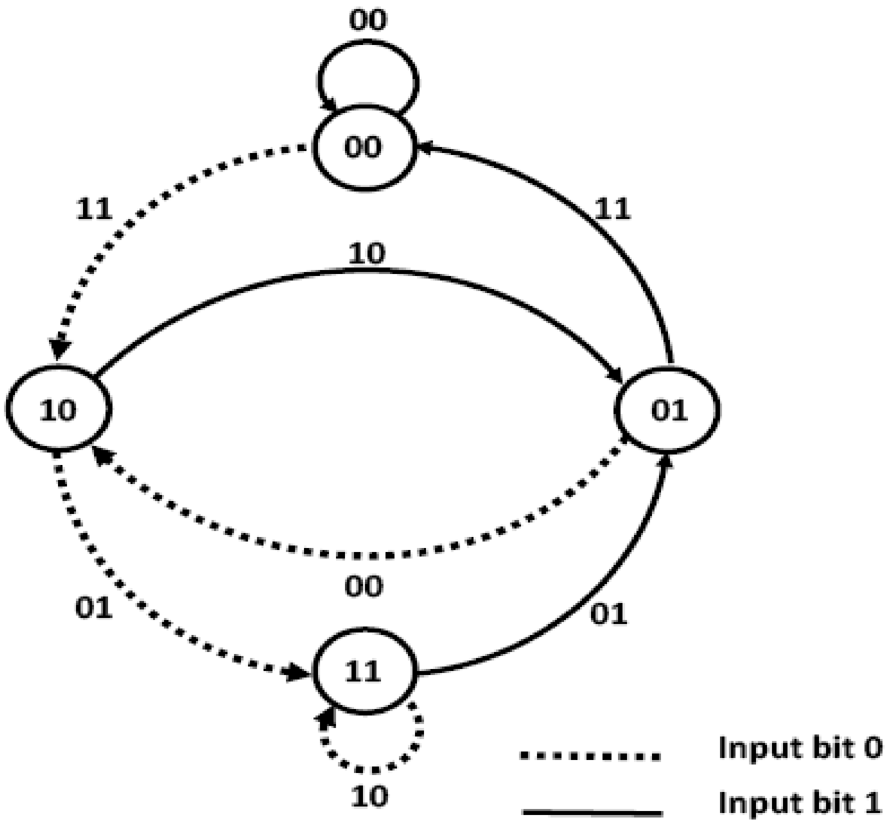


Figure 2-2: State Diagram of a Convolutional Encoder with K=3, k/n=1/2 [1]

2.2 Viterbi Decoding

Viterbi Algorithm employs ***Maximum Likelihood Detection*** technique for decoding the convolutionally encoded data. The process involves formation of a ***'Trellis'*** structure which depicts a time-indexed information of the state diagram of the convolutional encoder. The process initiates at state **'00'**. As shown in the state diagram, there are **'k'** possible paths originating from each state, one for each kind of input bit (In this case: for 0 and 1 as k=2). Viterbi looks for the possible transitions from each state, from the state diagram. From each state, an edge follows for each kind of possible input and carries the ***distance*** which is a ***measure of similarity*** (error) between the reference outputs (labeled in the state diagram) and the bits received. This distance is called **'Branch Metric'** and provides a basis for retrieving the encoded bits. The branch metric values of all the connected nodes (that form a single path) are accumulated to generate a **'State Metric'** value at each terminating node [1].

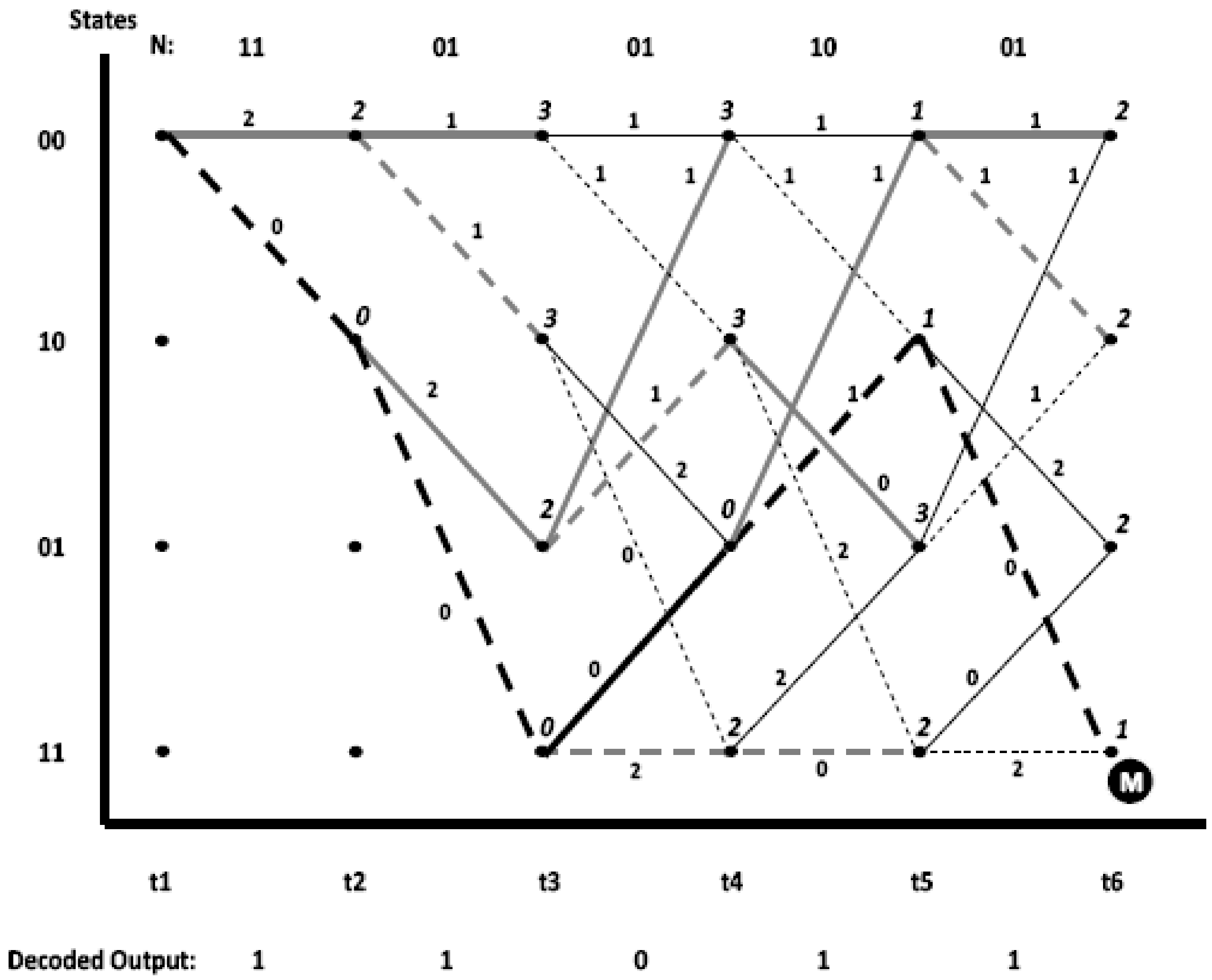


Figure 2-3: Formation of trellis and process of traceback for K=3 k/n=1/2 [1]

The bits are processed in groups depending on the code rate and so in this particular case, they will be processed in pairs. Figure-2-3 displays a six-stage trellis for the state diagram in Figure-2-2. As shown, at every new stage, the next pair is processed and distance is evaluated. Now, as each state can be entered through two possible paths, one path with the best metric has to be chosen. Such a path is called the ***surviving path*** and the best metric decision may be a ***minimum distance metric*** or a ***maximum likelihood metric***. Here for the sake of simplicity we have shown an example trellis with minimum distance metric and so a surviving path is the one with the minimum state-metric value. As different surviving paths exit within a trellis, they are shown with the help of bold-gray lines, whereas the final survivor path is shown in bold-black line. The dotted ones represent the branches for input 1 and solid lines represent transitions in case of input 0. The branch metrics are calculated by performing XOR-addition operation on the corresponding input pairs with all the reference outputs and are shown on all branches (Figure-2-3). State Metrics are shown at each node.

2.2.1 Trace Back

The trace back unit of the Viterbi decoder is of special interest. After the construction of trellis, it is traced-back from the last stage, based on the kind of the best metric criterion decided. The last stage of the trellis is searched for the best metric value (minimum distance or maximum likelihood). The node with the best metric value then acts as the root and the survivor path connecting to this root is selected for decoding the values. As the Figure-2-3 shows, the minimum state-metric value is ‘1’ and hence the node marked ***‘M’*** forms the root of the surviving path. The trellis maintains a memory of each branch's source and destination nodes and the expected input value at this branch (represented by dotted and solid lines). This memory helps the formation of the surviving path and the extraction of decoded sequence. E.g. the record says that the predecessor of M is at State $(10)_2$ - Stage 5 and as for each dotted line we record an input value ‘1’ and for each solid line, the decoded value is ‘0’, the first decoded value is 0. As tracing back is a reverse process so the first bit is decoded in the end.

In this way the whole input stream is extracted from the trellis. This resultant stream may or may not be the exact replica of the original input stream depending on the percentage of noise (SNR) induced into the encoded data by the channel. However, as studies reveal, Viterbi proves to be superior in performance over other decoding schemes, working in the same scenario.

2.3 Demodulator Configurations: Hard Decisions vs. Soft Decisions

After the encoding process has been applied on the input stream, the encoded bit stream needs to be modulated, before injecting it into the channel. Modulating the stream (code symbols) converts it into signal waveforms. The modulation schemes applied may be baseband or band pass. Whatever, the kind of modulation may be, the general rule is to map l code symbols at a time, to a signal waveform chosen from a set $M=2^l$ possible combinations. These waveforms are then transmitted through the channel where they are affected rather corrupted by the channel noise (AWGN in this particular case). The demodulator unit on the receiver side de-maps the waveforms back into a bit stream. This bit stream is then passed on to the decoding module.

Two kinds of configurations are possible for the demodulator output:

- ***Hard Decision***

  One of the ways compares the incoming waveforms with a certain threshold value. If the signal level appears to be higher than that threshold a bit value 1 is assumed otherwise a 0 is inferred. This configuration scheme which assumes a binary decision i.e. quantizes demodulator output to two levels is known as hard decoding/decision.
- ***Soft Decision***

  The other method of extracting the bit stream from the waveforms is called soft decoding or decision. This configuration compares the signal's level with more than one threshold values i.e. the decision is not binary. Hence the demodulator output is quantized to more than two levels i.e. more than 1 bit will be required for storing the different possible values. For example, a typical soft decoding system employing an 8 level (or more) de-mapping will be needing 3 bits for each signal level causing an increased requirement for memory space.

2.3.1 Hard Decision and Soft Decision Plane

Figure- 2-4(a) and 2-4(b) show the Hard and Soft decision planes respectively. As shown, the four hard combinations can be placed at the corners of a square. If any kind of noise is introduced in the transmitted data, any single bit change will cause the decision to move from one corner to another. When decoding it with the help of Viterbi's trellis, the distance (differences) are calculated as no of bits in error and is called ***Hamming distance***.

For Soft Viterbi decoding, the quantization levels between the corner values are increased. These corner values can be now converted to some voltage levels , like those in Figure -2-4(b), taken from the domain [-15,15]. Now, any kind of disturbance in the signal level cannot be calculated using Hamming distances. A preferable choice of metric may be Euclidean distance or dot product. The Figure-2-4(b) represents the distance as dot products between the corresponding values (the reference values at the corners and the coordinates of the actual point).

Figure-2-4(c) and 2-4(d) show the trellis for soft viterbi decoding. Figure-2-4(c) shows the reference values for each input (1 and 0) while 2-4(d) represents the distances as dot products.

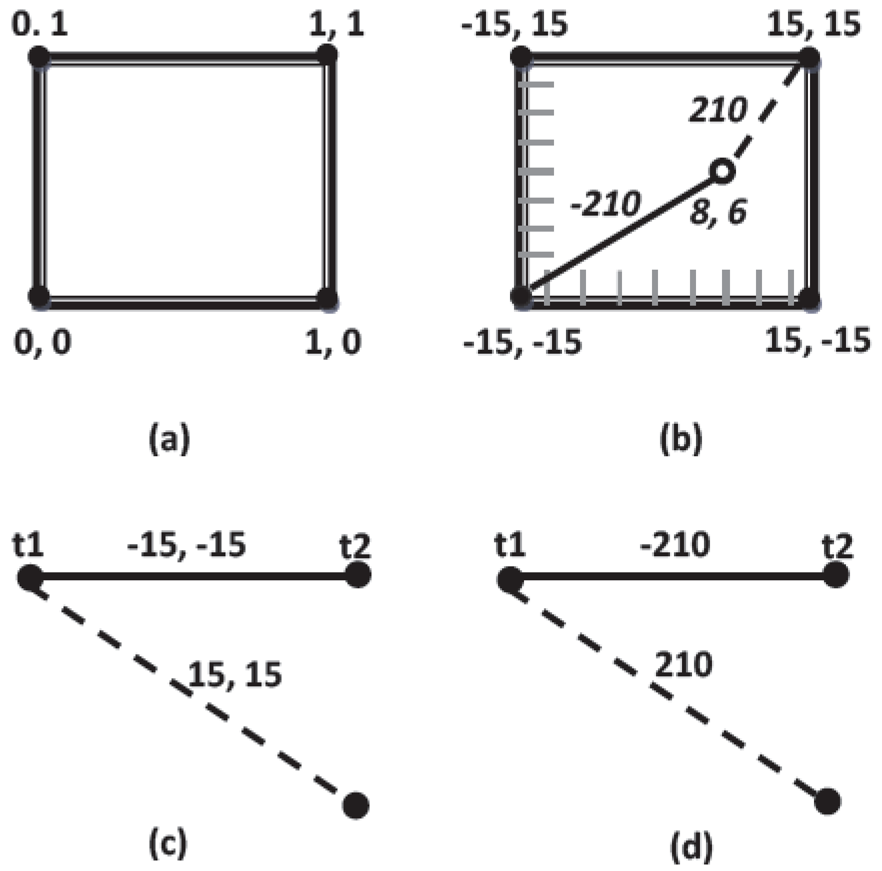


Figure 2-4: (a) Hard Decision Plane (b) Soft Decision Plane (c) Soft Viterbi Trellis with Reference values (d) Soft Viterbi Trellis showing distances (errors) in transmission as Dot Products [1]

There are some obvious pros and cons, associated with both these configuration schemes. Where hard decoding implies a simpler approach, the accuracy of the decoded result is doubted especially in case of low SNR. On the contrary, the soft decoding schemes show greater accuracy as the decisions tend to be really narrowed within small regions. Therefore soft decoding is employed largely in the communication systems where the accuracy of the received data is a critical factor. Hard decoding schemes may be applied for small firmware implementations where the complexity, associated with the soft decisions, needs to be avoided.

CHAPTER – 3 : HARDWARE PLATFORM

ADSP-BF533 Blackfin is an inexpensive RISC DSP, whose cycle time is 1.6ns (600MHz). The ADSP-BF533, ADSP-BF532, and ADSP-BF531 processors are enhanced members of the Blackfin processor family that offer significantly higher performance and lower power than previous Blackfin processors while retaining their ease-of-use and code compatibility benefits. These three processors are completely pin compatible, differing only in their performance and on-chip memory, mitigating many risks associated with new product development.

The Blackfin processor core architecture combines a dual MAC signal processing engine, an orthogonal RISC-like microprocessor instruction set, flexible Single Instruction, Multiple Data (SIMD) capabilities, and multimedia features into a single instruction set architecture.

Blackfin products feature dynamic power management. The ability to vary both the voltage and frequency of operation optimizes the power consumption profile to the specific task.

3.1 ADSP-BF533 Core Architecture

The ADSP-BF533 core consists of two 16-bit multipliers, two 40-bit accumulators, two 40-bit arithmetic logic units (ALUs), four 8-bit video ALUs, and a 40-bit shifter. These computational units are able to process 8, 16, or 32-bit data extracted from the register file.

3.1.1 Compute Register File

There are eight 32-bit registers comprising the compute register file. For 16 bit operations, these registers act as sixteen 16-bit registers.

3.1.2 Arithmetic and Logical Unit -- ALUs

The two 40 bit ALUs operate on 16, 32 and 40 bit operands. Fixed point addition and subtraction operations are supported. Addition and subtraction of immediate values, Accumulation and subtraction of multiplier results, Logical operations and bitwise operations, functions like ABS, MAX, MIN, Round, division primitives, conditional instructions are also supported.

Signed and unsigned formats, rounding, and saturation are supported.

3.1.3 Multiply and Accumulate -- MAC

Each MAC can perform fixed point multiplication and multiply-and accumulate operations on 16-bit fixed point operands resulting 32-bit or 40-bit values. Multiplication, Multiply-and-accumulate with addition, Multiply-and-accumulate with subtraction and dual versions of the two are supported. Saturation and Optional rounding formats are supported.

A MAC operation takes a single cycle for execution.

3.1.4 Shifter

The 40-bit shifter supports shifting, rotating, normalization, and extraction operations.

3.1.5 Zero Overhead Loops

Zero overhead looping is supported by the hardware. Moreover the architecture is fully interlocked, meaning there are no visible pipeline effects when executing instructions with data dependencies.

3.1.6 Address Arithmetic Unit

The address arithmetic unit provides two addresses for simultaneous dual fetches from memory. Eight 32-bit pointer registers are provided for C style stack manipulation.

3.2.6.1 Circular Buffering

A multiported register file consisting of four sets of 32-bit Index, Modify, Length, and Base registers is provided to support circular buffering. The ***Index (I)*** registers are used to hold the address that is sent out on the address bus. ***Base (B)*** registers contain the starting address of the circular buffer. ***Length (L)*** registers specify the length of the buffer while the ***Modify (M)*** registers contain the value (positive or negative) that will be added to the L registers at the end of each memory access. The size of the modify value must reside the length of the circular buffer.

The trellis unit can be efficiently benefited from this feature as the same memory can be re-used for building the new trellis stages each time. Furthermore, the arrays those need to be circularly accessed iteratively, e.g. Reference output array, can be implemented as circular buffers.

3.1.7 Blackfin Memory

Blackfin processors support a modified Harvard architecture in combination with a hierarchical memory structure. Level 1 (L1) memories typically operate at the full processor speed with little or no latency. At the L1 level, the instruction memory holds instructions only. The two data memories hold data, and a dedicated scratchpad data memory stores stack and local variable information.

In addition, multiple L1 memory blocks are provided, which may be configured as a mix of SRAM and cache. The Memory Management Unit (MMU) provides memory protection for individual tasks that may be operating on the core and may protect system registers from unintended access.

3.1.8 Instruction Set [9]

The processor's assembly language employs a simple algebraic syntax and the architecture is optimized for use with the C compiler. Where, the 16-bit opcodes represent the most frequently used operations, some 32-bit multifunction instructions are also included to support complex DSP operations. Blackfin is neither Super Scalar nor VLIW but supports the issuance of multiple instructions within a single instruction with some limitations.

3.2.6.1 Specialized Instructions

To facilitate and accelerate various signal processing tasks, the instruction set is equipped with many special instructions like compare/select and vector search instructions. Such vector search operations are useful in searching arrays, a requisite operation in trace-back unit.

Special video instructions are also included with byte alignment and packing operations, 16-bit and 8-bit adds with clipping, 8-bit average operations, and 8-bit subtract/absolute value/accumulate (SAA) operations.

3.2.6.2 Instruction Modes

40-bit ALU operations support Single, Dual and Quad 16 bit operations and Single and Dual 32-bit operations. Such configurations allow parallel operations processing, though ADSP-BF533 is neither super-scalar nor VLIW.

3.2.6.3 Parallel Instructions

As discussed, Blackfin processor supports a limited multi-issue capability. The general rule is to issue some instructions in parallel within a single instruction so that the instruction size is exactly 64 bits. A 32-bit instruction can be issued in parallel with two 16 bit instructions or with a single 16 bit instruction followed by a 16-bit NOP instruction. Two 16-bit instructions can be issued in parallel with an added 32 bit MNOP instruction. However, these do not apply for all the instructions as there is a limited set of instructions that can be issued in parallel.

In this way many of the core resources can be utilized within a single instruction cycle.

CHAPTER – 4: OPTIMIZATION STRATEGY

Programmers have been striving at their best to achieve the best possible optimization levels where ever possible. [4] discusses some standard optimization techniques necessary for any DSP based application. Our work focuses on the optimized implementation of Viterbi. In this chapter, we state the optimization strategy employed in the work while explaining all the phases the work has gone through.

4.1 The Optimization Process

Usually the optimization process involves two main steps:

- ***Logic Optimization***
- ***Code Optimization***

4.1.1 Logic Optimization

The logic optimization part involves simplifying the algorithm, searching for and removing any kind of possible redundancies. While there are some very well known logic optimization techniques, Viterbi's structure also reveals some really interesting facts. The following text highlights these possibilities.

4.1.1.1 Exploiting the Butterfly:

Viterbi can be logically optimized by observing its butterfly structure which shows many optimization possibilities. Different techniques that can make good use of the structure are explained below.

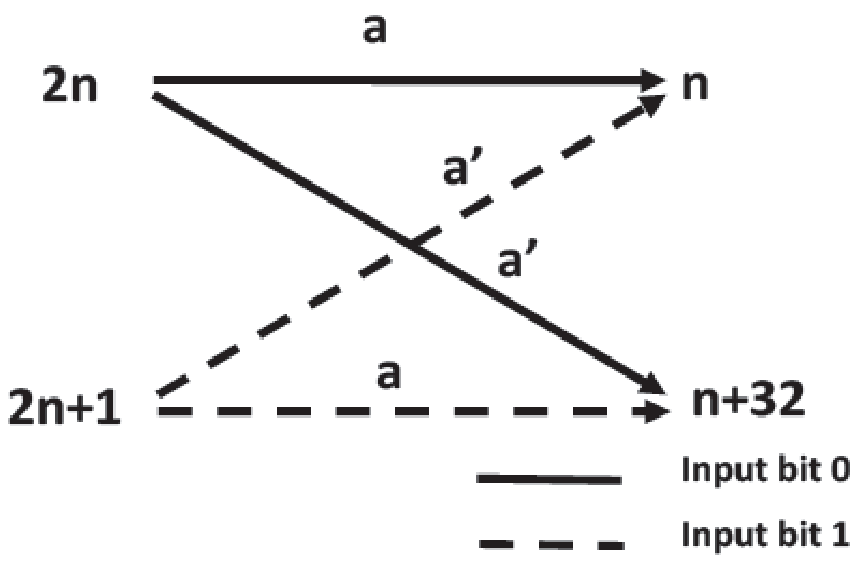


Figure 4-1: Symmetry in Viterbi's Butterfly

4.1.1.1.1 Loop unrolling:

Optimization may also be achieved by techniques like loop-unrolling, which reduces the loop iterations by repeatedly writing the part of code that needs to be looped. This effectively increases the execution speed at the cost of increased code-memory space requirements. As can be seen in Figure-2-3, each trellis stage requires a set of operations that need to be performed for each state, i.e. calculating metrics and deciding for the next optimal path. The butterfly structure however depicts a natural symmetry in the algorithm i.e. all the consecutive states can be paired as they show similar transitions. The required operations for the consecutive states are hence unwound so that the optimizer is permitted to find combinations of possible parallel instructions/operations. In this way a reduction by a factor of two can be observed in the number of loop iterations.

4.1.1.1.2 Branch Metric:

The butterfly reveals another interesting optimization possibility as shown in Figure-4-1, i.e. if one branch metric value say, ***'a'*** is known, the rest three are obvious. It means that out of four values we need to store only one reference value while the other three can be easily evaluated. In this way a total of 3/4th of the required memory space can be saved.

4.1.1.1.3 Elimination of transition memory:

Lastly, for the process of trace back, no transition memory is really required to be maintained as the butterfly symmetry clearly states that if the current node is ‘n’ or ‘n+32’ the previous states must be either ‘2*n’ or ‘2*n+1’ and so, with the addition of few simple operations, a great deal of memory space can be saved.

4.1.1.2 Use of Look-up Tables:

An important decision lies with the usage of if-else/switch structure and look-up tables. While the If-else/switch statements are simpler to code, they have an obvious disadvantage of blocking the pipeline. The if-else structure is treated as a branch instruction which cannot always be correctly predicted as a taken branch or not. The instructions following the branch instruction may or may not be executed depending on the condition being checked and thus it blocks the pipelined path.

The use of look-up tables has no such disadvantage. Instead, it helps increasing the execution speed but at the cost of increased data-memory requirements. The switch structure can be replaced by a look-up table by mapping all the cases as indices and placing all the resulting values into the arrays at the corresponding indices of the array.

4.1.1.3 Off-line Processing:

Look-up tables may also be employed to substitute such arithmetic and logical operations whose results are obvious / static. Results of certain operations can be pre-calculated and stored in such tables. This is called off-line processing.

For instance, in case of Hard Viterbi implementation, for each state and input a value of branch metric has to be calculated which requires an XOR (modulo-2 add) operation. This operation may take 3 to 4 clock cycles. However if we pre-calculate all the possible branch metrics and keep them in a look-up table, it will require a single memory read cycle. Hence a great deal of improvement in execution speed is observed.

4.1.1.4 Optimal MAC:

Where use of look-up tables is really advantageous in case of hard Viterbi decoding; for Soft Viterbi implementation, use of Look-up tables for keeping the branch metric values must be avoided. As discussed before, Soft decisions increase the number of different possible signal levels in the incoming encoded information. This requires an additional look-up table for mapping these signal levels to array indices so that the branch metric values placed in metric-look-up table can be accessed. This doubles the memory access time. Instead, a MAC (Multiply and Accumulate) operation requires less execution time as compared to the memory access time required for the looking-up process and so a better logic implementation can be achieved using MAC operation.

4.1.2 Code Optimization

Once a suitable level of optimization has been achieved logically, the code is analyzed at the architectural level to look for further optimization possibilities. This step is especially done as a part of hardware implementation. During this phase, an effort is made to exploit the core architecture to seek as many benefits as possible. The code optimization part can benefit from the following features and techniques [7].

4.1.2.1 Specialized Instructions:

Optimization at this level greatly depends on the instruction set architecture of the hardware platform in consideration (***DSP***, in our discussion). DSPs are equipped with a set of well tuned specialized instructions that form a flexible and densely encoded instruction set compiling to a very small memory size. Use of such specialized instructions in the code is really advantageous in producing an optimized product, in terms of both memory and speed. With the help of multi-functional instructions many of the processor resources may be used in a single instruction. MAC instruction is one of the examples of such specialized instructions. It executes the MAC operation i.e. Multiply/Accumulate in a single cycle. However if such an operation is done with the help of individual Multiply and Add instructions, it will take more than 1 cycle.

4.1.2.2 Parallel Operations:

Capability of executing operations in parallel is one of the most desirable features in any DSP processor as it helps achieving a higher level of optimization and thus largely improves the system's throughput. A wide range of DSP processors are available in market with varying features, designed for different kinds of applications; Super-scalar and VLIW architectures being a popular choice wherever a speedy system is targeted. These systems permit issuance of more than a single instruction, to the execution units, per cycle. Super-scalar machines can dynamically issue multiple instructions per clock cycle whereas VLIW Processors use a long instruction word that contains a usually fixed number of instructions that are fetched, decoded, issued, and executed synchronously. Though these architectures are really advantageous still they add to the complexity and cost of the system which cannot always be afforded. In such a situation an inexpensive simpler system, with added capability of allowing some instructions to be issued in parallel, like ADSP Blackfin, provides efficient implementations both in terms of cost and performance.

4.1.2.3 Loop Unrolling w.r.t Code:

Looping structures present in the code can be unrolled for improving the execution speed but, as discussed before, at the cost of increased code memory size. Unrolling the loop makes it possible to take maximum benefits of the core architecture. For example once the statements are unrolled, independent operations/processes may be carried out in parallel to each other.

4.1.2.4 Zero Overhead Loops:

Software loop overhead can be significantly reduced by assigning the control of the loop to the hardware. The loop counter register holds a value which is based on the test instructions. This counter is decremented and looping continues until the counter reaches zero. Such a strategy loads off the operating system from the overhead of testing loop condition every time (for each iteration) and thus a significant improvement in performance is observed.

CHAPTER – 5 : SYSTEM DESIGN

This section discusses the implementation details of our design on a DSP platform. Our aim is to design an optimized version of Viterbi having a constraint length K=7 i.e. for a total of 64 states with a code rate of 1/2. The size of input buffer is 10 samples and the trace back depth is set as 100.

5.1 System Model

Figure-5-1 shows the implemented system model. As shown, the digital input stream ***X*** is convolutionally encoded and the resultant (digital) stream ***Y*** is modulated/mapped onto different signal levels i.e. an analog signal ***Z*** is formed. This modulated signal is transmitted through the channel where it is corrupted by the channel noise (Additive White Gaussian Noise here).

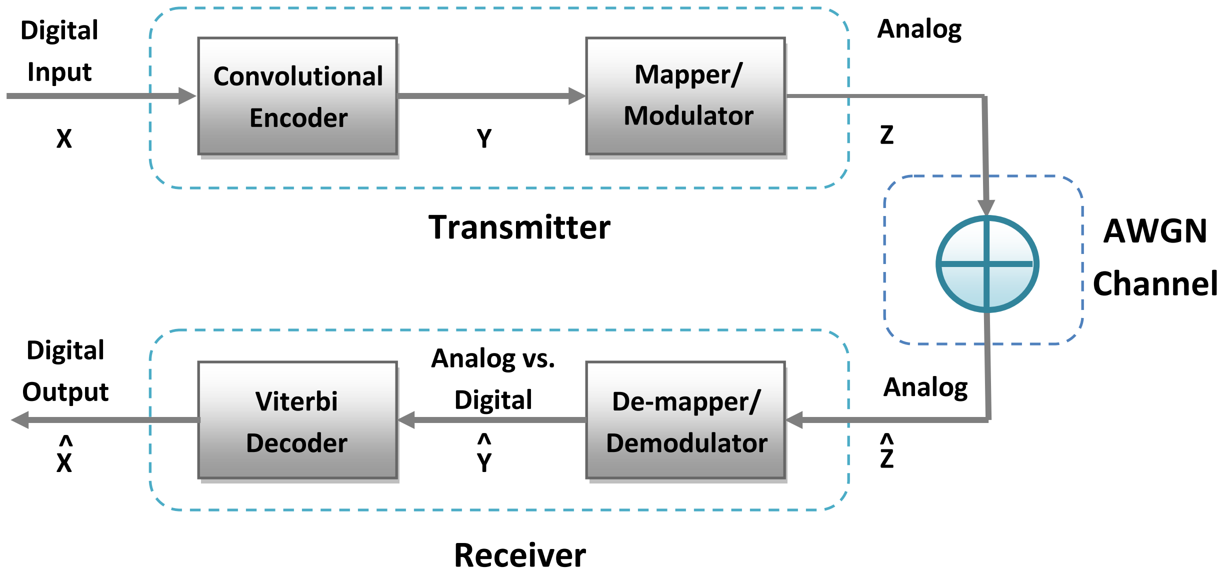


Figure 5-1: System Model

The corrupted signal Z **,** when received, is passed on to the de-mapper/demodulator. The demodulator's output, as discussed before, may be configured as analog or digital. If the decision is binary then a digital stream based on hard values is generated, otherwise analog values, based on soft decisions, are produced. Once the configuration has been decided, the demodulated stream $\hat{Y}$ is passed on to the corresponding Viterbi Decoder (Hard or Soft). The decoded output $\hat{X}$ can then be compared to the actual input of the transmitter block ***X*** to calculate the ***Bit Error Rate (BER)***.

5.1.1 Inside the Decoder

Figure-5-2 shows Decoder's internal design. As shown, once the input buffer is filled with the requisite amount of input samples, the data is sent out to the Trellis unit in form of chunks. Length of the buffer dictates the input latency but here we assume that the buffer is already filled whenever the data is required and so the input latency is zero. The Trellis unit builds up the trellis. The buffer keeps on feeding the trellis until the length of the trellis reaches the trace back depth. At this point, the trellis is traced back by the Trace Back unit to produce the decoded data.

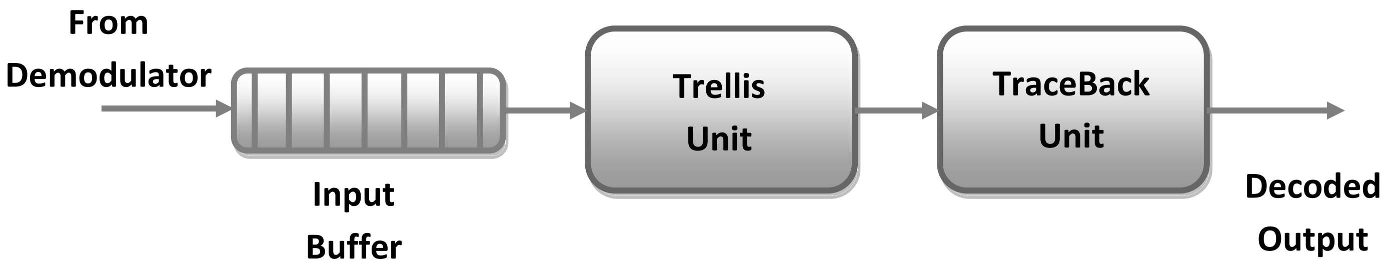
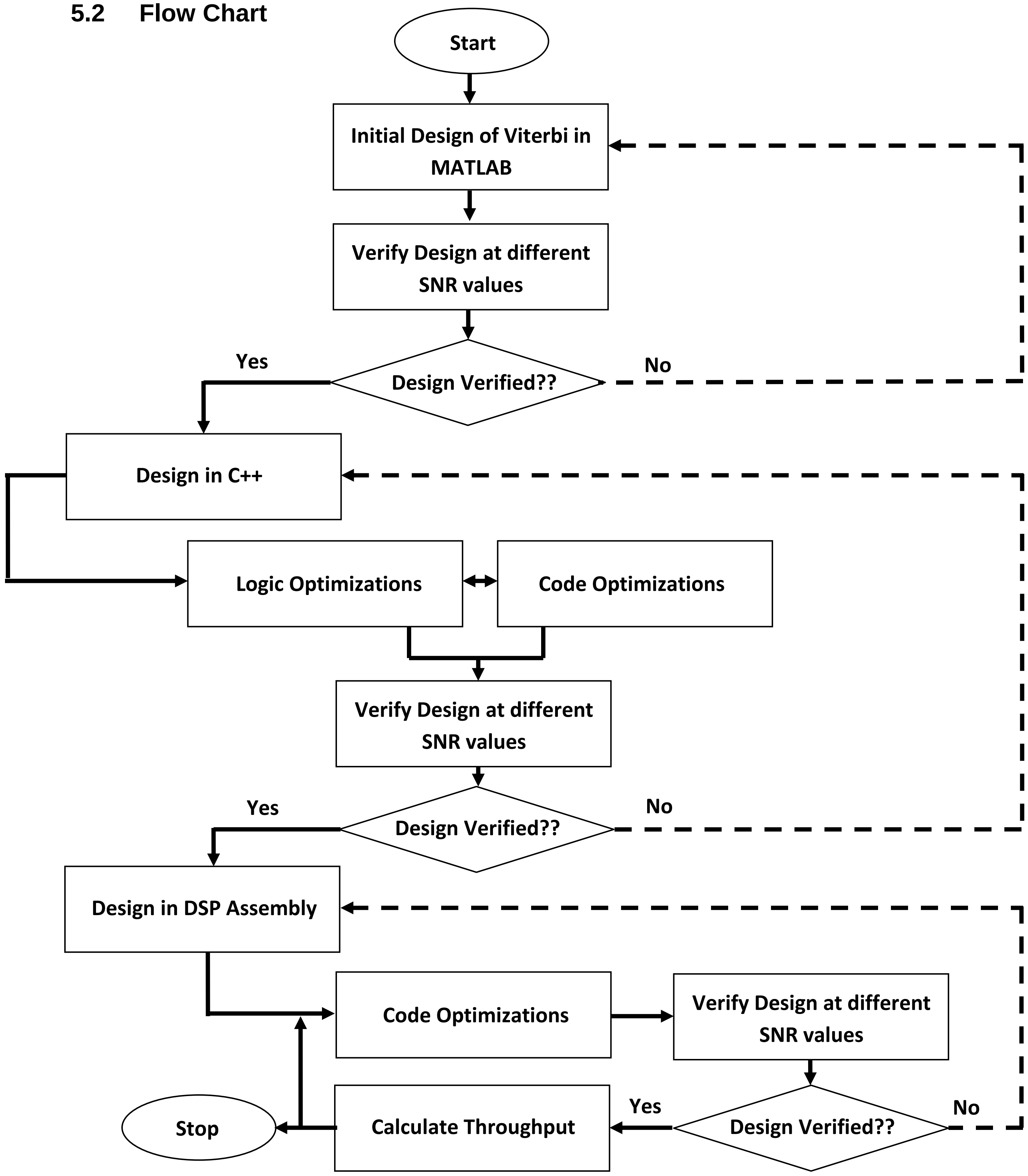


Figure 5-2: Decoder Configuration

Once the trellis is fully traced back, the next call to the trace back unit needs to be initiated immediately after single buffer fetch instead of waiting for the re-fill of the whole trellis. This significantly eliminates the latency required for re-filling all the trellis stages and reduces the processing time required for tracing back the whole trellis at once and thus increases the system’s average speed.

The trellis can be implemented with the help of circular buffer so that the same memory is re-used because once it is traced back the data occupying the trellis is no longer useful. This helps reducing a significant amount of required memory space.

5.2 Flow Chart



CHAPTER – 6 : EXPERIMENTAL RESULTS

This chapter covers the details of the experiments that were carried out for testing the designed system. The results of these experiments show a massive improvement in performance both in terms of speed and memory as explained in this chapter.

6.1 Methodology:

6.1.1 Phase I: MATLAB

MATLAB, a high performance technical language, proves to be a very handy tool for initiating any technical project. The toolboxes available in MATLAB provide a set of built in commands and functions that facilitate the designers to implement several complex modules in a simpler way. Our concern for our work has been the communication systems toolbox which provides with all the necessary modules required in our design. So, the first step of implementation involves the coding in MATLAB.

An initial design of the system can be easily developed in MATLAB and can be further analyzed and improved through simulations. The steps include:

i. The input to the system is a randomly generated bit stream (vector) of length 1000.

ii. This input is then convolutionally encoded (K=7, code rate=1/2) using the built-in function as discussed in previous chapter.

iii. The resultant encoded data has to be transmitted through the channel but before transmission it is mapped onto different signal levels. The encoded information is modulated through BPSK modulation scheme **Vpp = [-√2, +√2]**.

iv. After this mapping process has been finished, the signal is transmitted through AWGN channel. MATLAB has a pre-defined routine for AWGN channel. By passing the SNR and Signal Power values as arguments, any signal can be sent and thus corrupted through the channel. The Signal-to-Noise ratio (SNR) is continuously varied from minimum to maximum so that the system can be tested well for the whole SNR range.

v. The corrupted signal is then demodulated/de-mapped within the range [-15,+15] for soft decisions and [0,1] for hard decisions.

vi. The de-mapped data is sent to the Viterbi decoder module. This is where our interest lies. A great deal of logic optimization techniques may be applied over here to produce a first level efficient code. This code can then be compared to the available Vitdec function in MATLAB. A comparison of the resultant Bit error rates of our developed Hard Viterbi codes, Soft Viterbi code and the available Vitdec MATLAB function is provided in the next section.

6.1.2 Phase II: C++

Once a working system is developed on MATLAB and the results are verified, the next step is to initiate the optimization process for final implementation. Before planning for the hardware implementation and optimization, an intermediate design on C/C++ is important. Most of the available DSPs support the C/C++ codes so it is always a necessary step to code the system for the supported language i.e. C/C++ in our work.

A greater level of logical optimization can be achieved at this level as all the built-in routines used in MATLAB need to be expanded into their definitions in C/C++. These expansions and re-definitions allow us to introduce further improvements into the design.

Some code optimization techniques like loop unrolling may also be applied in this phase. The good thing about this intermediate product is that it can be taken directly, without much modification (which is required if some libraries are not supported), to the hardware platform.

6.1.3 Phase III: Final Design on DSP

The last phase of the implementation process is based on the work done on the hardware platform. This phase involves the application of code optimization techniques, which largely depends on the core architecture of the hardware. So, before the application of any such technique, a firm knowhow of the processor's architecture is required. Different DSPs with differing features are available in the market.

6.2 Bit Error Rate Calculation and Comparison:

As stated before, in the first step of building the code in MATLAB, the code has to be analyzed for the minimum (output) BER (at least less than or comparable to that of the existing ones).

The BER calculation is done for 10 SNR values. From the theory of Communication systems, it is a known fact that Bit Error Rate and SNR values are inversely proportional to each other i.e. BER will be maximum at lower SNR values while it will reach zero as the SNR approaches the maximum limit.

The Bit Error rate is computed using the following formula:

$$\frac{\textbf{No of bits in error in the decoded stream}}{\textbf{Total no of bits}}$$

The BER comparisons are also made for the code developed in C++ and Blackfin's assembly to check for the bit accuracy.

6.3.1.1 Results

Figure-6.1 shows a comparison of resultant average Bit Error Rates (for fifty executions) among the built-in MATLAB code for (Hard) Viterbi decoding -- ***Vitdec*** and our designed codes for Hard and Soft Viterbi decoding, and for the Un-coded bit stream, for different SNR values.

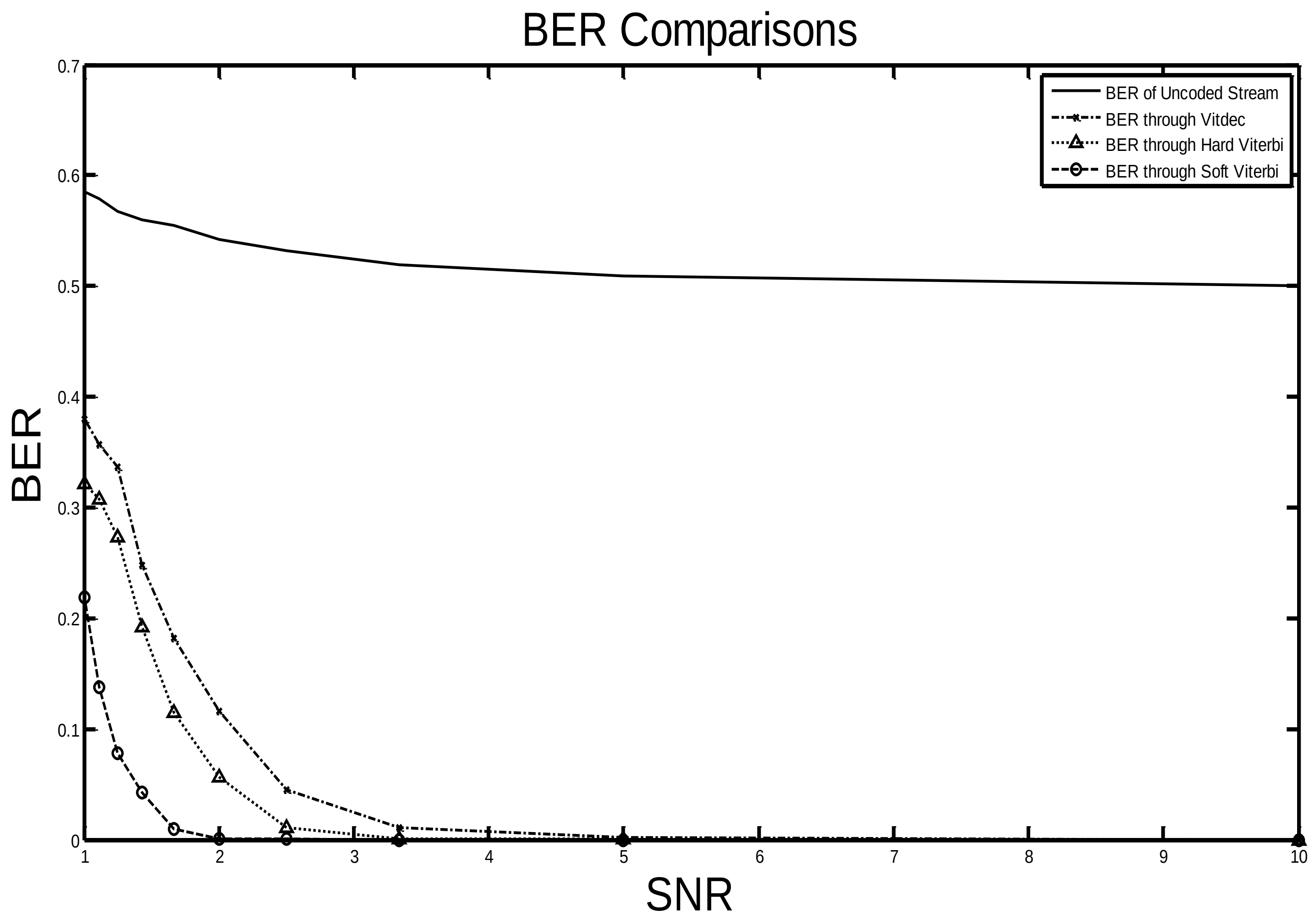


Figure 6-1: Bit Error Rate Comparisons

The Un-coded Bit stream obviously presents the worst case. The next curve shows the performance of Vitdec which is the next worse over here. Our hard Viterbi code shows a better performance over Vitdec while the Soft Viterbi decoder shows the best performance in terms of BER as the BER value approaches zero at such a small SNR value slightly more than ***'2'***.

6.3.1.1 Comparative Results on C++ and DSP

Once the code has been ported to C++ environment and that on DSP's, the bit-accuracy and performance need to be verified. For this reason Bit-Error-Rates, for the same SNR range, of the output (decoded) bits by the C++ code and Blackfin's assembly code have been presented in Figures 6.2 and 6.3.

As clear from these figures, all the three versions of Viterbi's design (MATLAB, C++ and DSP) show exactly the same performance characteristics, proving the bit-accuracy of the design.

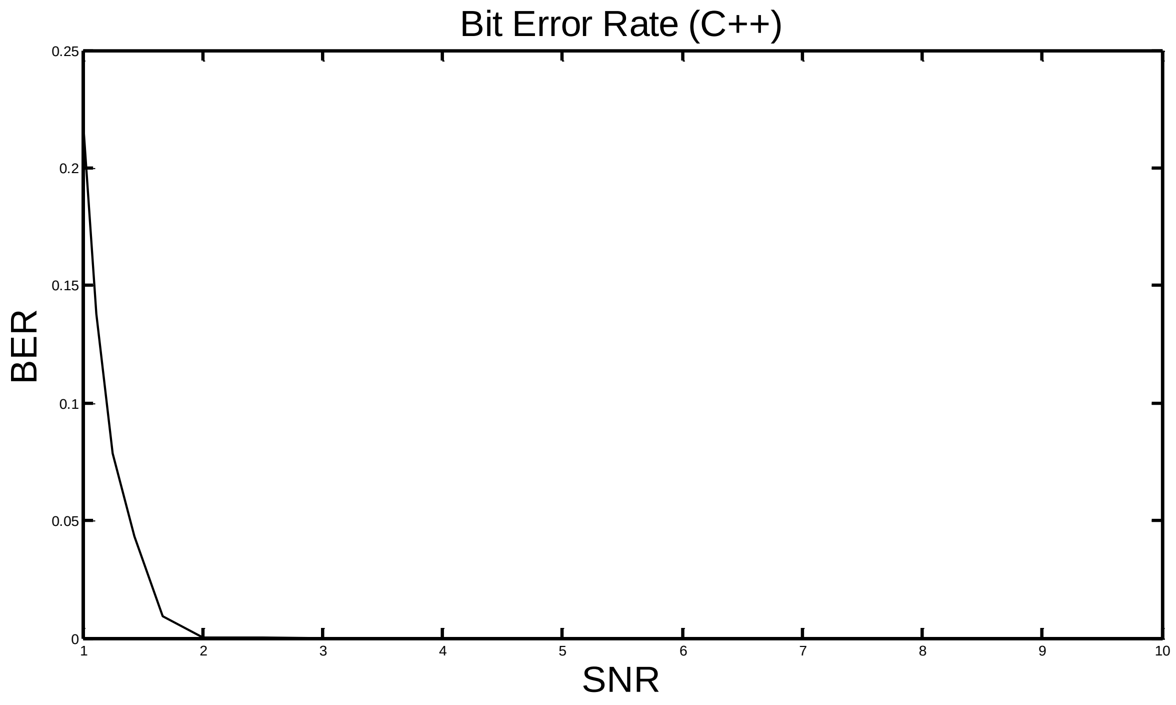


Figure 6-2: Bit Error Rate Comparison (C++)

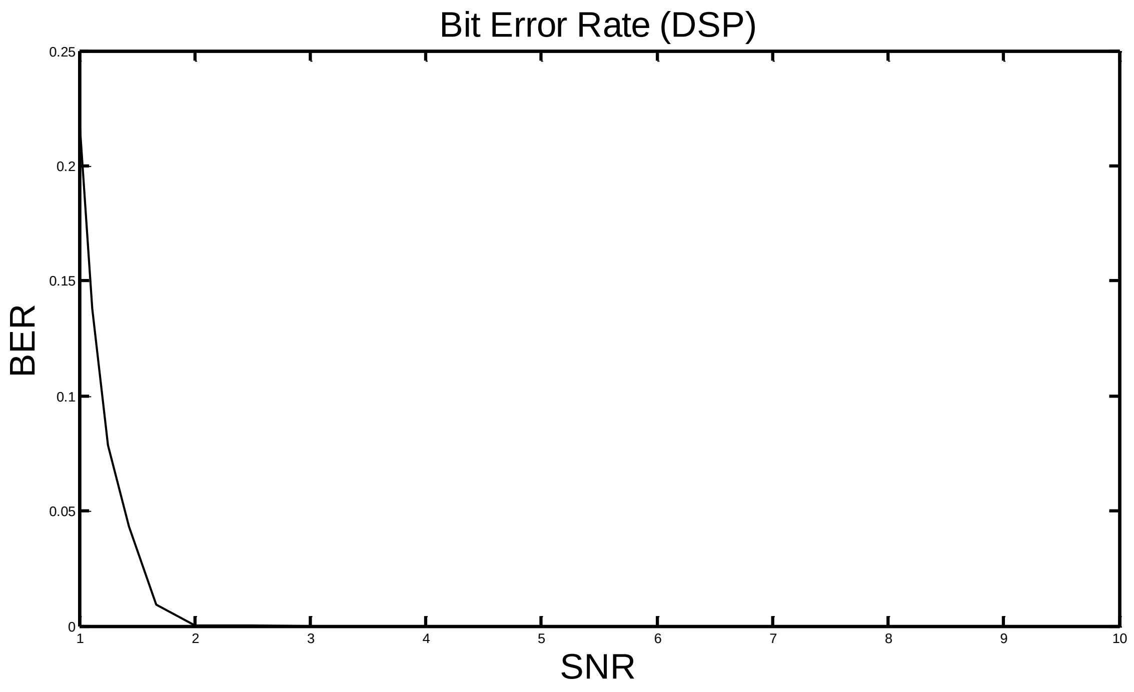


Figure 6-3: Bit Error Rate Comparison (DSP)

6.3 Benchmarking – System's Execution time and Throughput:

The designed code, after the application of logic and code optimization techniques, is taken to the hardware platform where an analysis, about its total execution time in terms of cycles (and so the throughput), has to be made.

6.3.1 ADSP BF-533 (For K=7 and code rate= 1/2)

The primary implementation has discussed before is on ADSP-BF533. For a constraint length equal to 7 with code rate=1/2, the single trellis call takes 266 execution cycles per bit. The traceback module takes 30.3 cycles per bit. The total execution time (including all calls to the trellis and traceback modules) is 334 cycles per bit.

The clock frequency is 600MHz and so the calculated throughput is

=600MHz/334 cycles-per-bit

=1.7Mbps

6.3.2 TMS320C6201 (For K=7 and code rate= 1/2)

Performance of the same design (K=7, code rate=1/2) is estimated for TMS320C620. As the clock frequency of TMS320C6201 is 200MHz, so the calculated throughput is

=200MHz/334 cycles-per-bit

=598kbps

6.3.3 ADSP-BF 533 (For K=9 and code rate= 1/2)

The third part of analysis is performed by increasing the constraint length from 7 to 9. If the constraint length increases the number loop iterations for forming the trellis also increases (as the number of states has increased). Now the trellis takes 1163.5 execution cycles-per-bit. The traceback module takes 43.74 cycles-per-bit and the overall execution time per bit is 1211.44 cycles.

The calculated throughput is

=600MHz/1211.44 cycles-per-bit

=495kbps

6.3.4 TMS320C6201 (For K=9 and code rate= 1/2)

Performance of the new design (K=7, code rate=1/2) is estimated for TMS320C620 as well. As the clock frequency of TMS320C6201 is 200MHz, so the calculated throughput is

=200MHz/1211.44 cycles-per-bit

=165kbps

These results are summarized in Table 6-1. Table 6-1 shows the benchmarking results after simulating the same code on different platforms with different clock frequencies and constraint length. The code rate however, has been kept constant i.e. 1/2. Another important point to be made is that the total no of cycles per bit contains not only the execution cycles required for Trellis and Trace-back Units but also includes the cycles consumed by the rest of the system.

As clear from the Table 6-1, the output data rate in case of BF533 is 1.7Mbps. If the same code is analyzed for a clock frequency of 200MHz like that of TMS320C6201 then reduction in data rate is obvious (Table 1). For a constraint length K=9, the data rate is further reduced to 495kbps (ADSP-BF533) and 165kbps (TMS320C6201-ignoring VLIW here).

Table 6-1: Benchmarking Results

Modules	**Cycles per bit For K=7 Clk Freq =600MHz**	**Cycles per bit For K=7 Clk Freq =200MHz**	**Cycles per bit For K=9 Clk Freq =600MHz**	**Cycles per bit For K=9 Clk Freq =200MHz**
Trellis	266	266	1163.5	1163.5
Traceback	30.3	30.3	43.74	43.74
Total	334	334	1211.44	1211.44
Data Rate	1.7Mbps	598kbps	495kbps	165kbps

6.3.5 Comparison with previously published work

G. Kang presents his work on a K=9 Viterbi with a code rate = 1/3 implemented on a **TMS320C6201 DSP** with clock frequency= 200MHz [3]. Table 6.2 shows an estimate of the performance on this same scenario and compares it with that of the Kang's claim.

Table 6-2: Comparison with other researchers

	Our Estimate for the same scenario	**Kang's Result**
Data Rate	165kbps	88kbps

As clear, the presented optimized code provides a data rate which is almost 2 times to that claimed by the researchers in [3].

CHAPTER – 7 : CONCLUSION AND FUTURE WORK

This work describes a generalized implementation of Viterbi decoder that has been optimized for a DSP platform. Different optimization techniques have been discussed including the ones that optimize the logic and those which optimize the code at the architectural level. Such techniques enabled to present a product which can be implemented on any DSP platform. For a 1/2 code rate we have implemented the design on ADSP-BF533 and estimated its performance for TMS320C6201 DSP as well. With K=7, an output data rate of 1.7Mbps on ADSP-BF533 (600MHz) has been achieved whereas 598kbps throughput has been estimated for TMS320C6201 DSP (200MHz). If the constraint length K=9 is considered then this data rate reduces to 165kbps on TMS320C6201 DSP. Briefly, an extremely optimized system both in terms of memory and execution speed has been developed. This system can be employed as a decoding unit in any CDMA/WCDMA communication system.

The embedded application designing is a vast and open field of research always welcoming innovative ideas because the lust for speed will continue forever. Particularly for Viterbi's implementation, some research can be done for exploring the influence of different factors like input buffer length, trace-back depth and inclusion/exclusion of transition memory from the design on the optimization of the system.

REFERENCES :

[1] B. SKLAR, DIGITAL COMMUNICATIONS, FUNDAMENTALS AND APPLICATIONS, 2ND ED. PEARSON EDUCATION, 2006.

[2] G. DAVID FORNEY, JR., THE VITERBI ALGORITHM: A PERSONAL HISTORY, PRESENTED AT THE VITERBI CONFERENCE, UNIVERSITY OF SOUTHERN CALIFORNIA, LOS ANGELES, MARCH 8, 2005.

[3] G. KANG, P. ZHANG., THE IMPLEMENTATION OF VITERBI DECODER ON TMS320C6201 DSP IN W-CDMA SYSTEM, COMMUNICATION TECHNOLOGY PROCEEDINGS, 2000. WCC - ICCT 2000. INTERNATIONAL CONFERENCE ON, VOLUME 2, ISSUE , 2000 PAGE(S):1693 - 1696 VOL.2.

[4] G. C. AHLQUIST, M. RICE, AND B. NELSON., ERROR CONTROL CODING IN SOFTWARE RADIOS: AN FPGA APPROACH, IEEE PERSONAL COMMUNICATIONS, MAG., AUG. 1999, PP. 35-39.

[5] E.Y.S AUGSBURGER, WM. RHETT DAVIS, B.NIKOLI., 500 MB/S SOFT OUTPUT VITERBI DECODER, SOLID-STATE CIRCUITS CONFERENCE, 2002. ESSCIRC 2002. PROCEEDINGS OF THE 28TH EUROPEAN VOLUME , ISSUE , 24-26 SEPT. 2002 PAGE(S): 523 - 526

[6] BUPESH PANDITA, SUBIR K ROY., DESIGN AND IMPLEMENTATION OF A VITERBI DECODER USING FPGAS, VLSI DESIGN, 1999. PROCEEDINGS. TWELFTH INTERNATIONAL CONFERENCE ON VOLUME , ISSUE , 7-10 JAN 1999 PAGE(S):611 - 614.

[7] MIT DEPARTMENT OF EECS, CAMBRIDGE, MA., CODE OPTIMIZATION TECHNIQUES FOR EMBEDDED DSP MICROPROCESSORS, DESIGN AUTOMATION, 1995. DAC APOS; 95. 32ND CONFERENCE ON VOLUME , ISSUE , 1995 PAGE(S):599 - 604

[8] ADSP-BF533 BLACKFIN PROCESSOR HARDWARE REFERENCE, ANALOG DEVICES INC, JULY 2006.

[9] ADSP-BF53X BLACKFIN PROCESSOR INSTRUCTION SET REFERENCE, ANALOG DEVICES INC, MAY 2003.

[10] TMS320C6000 CPU AND INSTRUCTION SET USERS GUIDE, TEXAS INSTRUMENTS CORPORATION, JULY 2006, PP. 1-4.

APPENDIX-A

CODES

A.1 MATLAB

Soft Viterbi:

```
function d = softvitty (code, numInputSymbols, …
numOutputSymbols, numStates, nextStates, outputs)


% Look up table for keeping the Signal Power values (soft)
softout=[-15,-15;-15,15;15,-15;15,15];


% Initializing 1st Column of trellis
State_metric(1,1)=4500;
State_metric(2:64,1)=-4500;


d=[ ];
k=1;


% Building the trellis
for  i=1:2:length(code)
    for j=0:2:63
```

```
        n1=floor(j/2);

        n2=n1+32;


        State_metric(n1+1,k+1)=max(State_metric(j+1,k)+

        mac(code(i),code(i+1),softout(outputs(j+1,1)+1,1),softout(outputs(j+1,1)+1,2)),

        State_metric(j+2,k)+mac(code(i),code(i+1),softout(outputs(j+2,1)+1,1),softout(o
        utputs(j+2,1)+1,2)));

        State_metric(n2+1,k+1)=max(State_metric(j+1,k)+

        mac(code(i),code(i+1),softout(outputs(j+2,1)+1,1),softout(outputs(j+2,1)+1,2)),

        State_metric(j+2,k)+mac(code(i),code(i+1),softout(outputs(j+1,1)+1,1),softout(o
        utputs(j+1,1)+1,2)));

    end

    k=k+1;

end


% Searching for the node with the best metric (Maximum Likelihood here)

mini=find(State_metric(:,k)==max(State_metric(:,k)));

mini=mini-1;


% Trace Back

        for i=length(State_metric)-1:-1:1

            d=[floor(mini(1)/32) d]
            mini=bitand(mini,31)*2
            if(State_metric(mini+1,i)>State_metric(mini+2,i))
               mini=mini;
            else
               mini=mini+1;
            end
         end
```

Hard Viterbi (No Transition Memory):

```
function d=vitty(code,numInputSymbols ,…
numOutputSymbols,numStates,nextStates,outputs)


% Lookup Table for keeping Branch Metric Values
Branch_Metric=[0,1,1,2;1,0,2,1;1,2,0,1;2,1,1,0];
State_metric=[];


% Initializing the 1st Column of Trellis (Best Metric: Minimum Likelihood)
State_metric(1,1)=0;
State_metric(2:64,1)=200;
d=[ ];
```

```
% Building the Trellis

for i=1:length(code)

   for j=0:2:63

      n1=floor(j/2);

      n2=n1+32;

     %%%%%%% Calculating State Metric %%%%%%%%%%%%%%%%%

     State_metric(n1+1,i+1)  =  min(State_metric(j+1,i)+

     Branch_Metric(code(i)+1,outputs(j+1,1)+1),

     State_metric(j+2,i)+ Branch_Metric(code(i)+1,outputs(j+1,2)+1));



     State_metric(n2+1,i+1)  =  min(State_metric(j+1,i)+

     Branch_Metric(code(i)+1,outputs(j+1,2)+1),

     State_metric(j+2,i)+ Branch_Metric(code(i)+1,outputs(j+1,1)+1));

   end

end


mini=find(State_metric(:,i+1)==min(State_metric(:,i+1)));

mini=mini-1;
```

```
%%%%%%% Trace Back %%%%%%%%%%%%%%%%%


      for i=length(code):-1:1

         d=[floor(mini(1)/32) d];

         mini=bitand(mini,31)*2;

         if(State_metric(mini+1,i)<State_metric(mini+2,i))

            mini=mini;

         else

            mini=mini+1;

         end

     end
```

Hard Viterbi (Including Transition Memory):

```
function d=vitty2(code,numInputSymbols ,…
numOutputSymbols,numStates,nextStates,outputs)


% Lookup Table for keeping Branch Metric Values
Branch_Metric=[0,1,1,2;1,0,2,1;1,2,0,1;2,1,1,0];
State_metric=[];


% Initializing the 1st Column of Trellis (Best Metric: Minimum Likelihood)
State_metric(1,1)=0;
State_metric(2:64,1)=200;

d=[ ];

index=State_metric;


% Building the Trellis

for i=1:length(code)

    for j=0:2:63

       n1=floor(j/2);

       n2=n1+32;

%%%%%% Keep the track of the best metric path %%%%%%%%%%%%%%

        if ( State_metric(j+1,i)+Branch_Metric(code(i)+1,outputs(j+1,1)+1)

          <State_metric(j+2,i)+Branch_Metric(code(i)+1,outputs(j+1,2)+1) )

                     index(n1+1,i+1)=j;

        else

                     index(n1+1,i+1)=j+1;                   end

         if( State_metric(j+1,i)+Branch_Metric(code(i)+1,outputs(j+1,2)+1)

        <State_metric(j+2,i)+Branch_Metric(code(i)+1,outputs(j+1,1)+1) )

                      index(n2+1,i+1)=j;

        else

                     index(n2+1,i+1)=j+1;

        end
```

```
     %%%%%%%% Calculating State Metric %%%%%%%%%%%%%%%%%%

      State_metric(n1+1,i+1)  =  min(State_metric(j+1,i)+

      Branch_Metric(code(i)+1,outputs(j+1,1)+1),

      State_metric(j+2,i)+ Branch_Metric(code(i)+1,outputs(j+1,2)+1));


      State_metric(n2+1,i+1)  =  min(State_metric(j+1,i)+

      Branch_Metric(code(i)+1,outputs(j+1,2)+1),

      State_metric(j+2,i)+ Branch_Metric(code(i)+1,outputs(j+1,1)+1));

   end

end

mini=find(State_metric(:,i+1)==min(State_metric(:,i+1)));

mini=mini-1;

%%%%%%%% Trace Back %%%%%%%%%%%%%%%%%%

       for i=length(code):-1:1

          d=[ floor(mini(1)/32)  d];

          mini=index(mini(1)+1,i+1);

       end
```

MAC:

```
function rslt=mac(a,b,c,d);

rslt=a*c+b*d;
```

Modulation:

```
function out= Modulate(in)
for i=1:length(in)
    if(in(i)==1)
        out(i)=15;
    else
        out(i)=-15;
    end
end
```

Demodulate:

```
function out= demodulate(in)
for i=1:length(in)
    if(in(i)<=0)
        out(i)=0;
    else
        out(i)=1;
    end
end
```

Hard Viterbi Calling Routine:

```
function d= vittest(code,t);

code_str=[];

for k=1:t.numInputSymbols:length(code)

   c=code(1,k:k+t.numInputSymbols-1);

   m=num2str(c(1));

   for l=2:length(c)

      m=strcat (m,num2str(c(l)));

   end

   code_str=[code_str; m];

end

code1=(bin2dec(code_str))';

d=vitty(code1,t.numInputSymbols,t.numOutputSymbols,t.numStates,t.nextStat
es,t.outputs);

d2=vitty2(code1,t.numInputSymbols,t.numOutputSymbols,t.numStates,t.nextSt
ates,t.outputs);
```

Main Function:

```
%%%%% Random Input %%%%%%

msg = mod(ceil(abs(randn(1,200))),2)


%%%%% Converting convolutional code polynomial to trellis description

t=poly2trellis(7,[117 155]);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%  numInputSymbols: 2

%  numOutputSymbols: 4

%  numStates: 64

%  nextStates: [64x2 double]

%  outputs: [64x2 double]

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%%%%% Passing Through convolutional Encoder %%%%%%%%%

code=convenc(msg,t);


%%%%%%%%% QPSK modulation %%%%%%%%

input=QPSK(code);

input2=QPSK(msg)   %% Uncoded Message
```

```
%%%%%%% These vectors will hold the BER values %%%%%%%%

acc=[];

vit_ac=[];

sft_ac=[];

ber_b4=[];

vitdec_avg=zeros(1,11);

vithard_avg=zeros(1,11);

ber_b4_avg=zeros(1,11);

softvit_avg=zeros(1,11);


%%%%%%% Calculate Signal Power %%%%%%%%

sigpower=(sum(input.^2))/length(input);


for i=1:50

   SNR=[];

   acc=[];

   vit_ac=[];

   sft_ac=[];

   ber_b4=[];
```

```
for cal=1:10:100


   %%%%%% Corrupting the code %%%%%%%%%

   npower=cal/100*sigpower;

   SNR=[SNR  sigpower/npower];


   %%%%%%% Corrupting the soft coded values %%%%%%%%%

   corrupted=floor( awgn (input,sigpower/npower,sigpower,'linear'));

   uncoded=floor ( awgn(input2,sigpower/npower,sigpower,'linear'));


   %%%%%%% Demodulation %%%%%%%%%

   hard_input=demodulate(corrupted);

   rec_uncoded=demodulate(uncoded);


   %%%%%%% Call Vitdec %%%%%%%%%%%%%%%%%%%%%%%%

   d1=vitdec(hard_input,t,2,'trunc','hard');


   %%%%%%%%%%%% Calling Hard Viterbi programs %%%%%%%%%

   d=vittest(hard_input,t);

   %%%%%%%%%%%% Soft Viterbi %%%%%%%%%%%%%%%%%%%%%%

   d3=softvitty(corrupted,t.numInputSymbols, …
   t.numOutputSymbols,t.numStates,t.nextStates,t.outputs)


   %%% Comparison of the decoded values with the original input msg %%%%

   vit_ac_calc=d1==msg;

   acc_calc=d==msg;

   acc_calc2=d2==msg;

   sft_ac_calc=d3==msg

   ber_b4_decod=rec_uncoded==msg;


   %%%%%%%%% Accumulating BER result  %%%%%%%%%%%%%%%%

   acc=[acc ((length(msg)-sum(acc_calc))/length(msg))];

   vit_ac=[vit_ac ((length(msg)-sum(vit_ac_calc))/length(msg))];

   acc2=[acc2 ((length(msg)-sum(acc_calc2))/length(msg))];

   sft_ac=[sft_ac ((length(msg)-sum(sft_ac_calc))/length(msg))];

   ber_b4=[ber_b4 ((length(code)-sum(ber_b4_decod))/length(code))];

end

vitdec_avg=vitdec_avg+vit_ac;

vithard_avg=vithard_avg+acc;

vithard2_avg=vithard2_avg+acc2;

softvit_avg=softvit_avg+sft_ac;
ber_b4_avg=ber_b4_avg+ber_b4;
end


vitdec_avg=vitdec_avg/i;
vithard_avg=vithard_avg/i;
vithard2_avg=vithard2_avg/i;
softvit_avg=softvit_avg/i;
ber_b4_avg=ber_b4_avg/i;


%%%%%%%%%% Plots %%%%%%%%%%%%%%%
plot(SNR,ber_b4_avg,'-o','Linewidth',2);
hold on
plot(SNR,vitdec_avg,'-.*','Linewidth',2);
plot(SNR,vithard_avg,':^','Linewidth',2);
plot(SNR,softvit_avg,'--s','Linewidth',2);
Xlabel('SNR','FontSize',25);
Ylabel('BER','FontSize',25);
Title('BER Comparisons','FontSize',25)
legend('BER of Uncoded Stream','BER through Vitdec','BER through Hard
Viterbi','BER through Soft Viterbi');
hold off;
```

A.2 C++

MAIN CODE:

```
#include"vit_data.h"

#include<stdio.h>

#include<math.h>

#include<iostream.h>

#include<conio.h>



const int MAX=101;

int msg_len=48;

int STATE_METRIC[64][101];

int siz=10;

int chunks=10;

int trace_back_depth=siz*chunks+1;

int ptr=0;

int d[100];

////////////////////////////// MAIN FUNCTION ///////////////////////////////////////

void main()

{

    int count;
int i=0,k=0;

   FILE *stream;

   stream=fopen("msg.bin","rb");

   if(stream!=NULL)

        count=fread(msgs,4,400,stream);

   STATE_METRIC[0][0]=4500;

   for(i=1;i<64;i++)

       {

                  STATE_METRIC[i][0]=-4500;

       }

  i=0;

  while(i<10)

       {

                  viterbi(i*siz);

                  i++;

       }

  traceback(d,0);

  while(i<(msg_len/siz))

                  {

                             viterbi(i*siz);

                             traceback(d,ptr-siz);
```

```
                    i++;


                }
fclose(stream);
}
```

/// VITERBI DECODER //

```
void viterbi(int k)
{
        int i=0;
        short n1;
        for(i=k;i<k+siz;i++)
        {
        for(int j=0;j<32;j++)
                {
                        n1=j*2;
STATE_METRIC[j][(ptr+1)%trace_back_depth]
= max(STATE_METRIC[n1][ptr]+mac(msg[i][0],msg[i][1],output[j][0],output[j][1]),
 STATE_METRIC[n1+1][ptr]+mac(msg[i][0],msg[i][1],-1*output[j][0],-1*output[j][1]));
 STATE_METRIC[j+32][(ptr+1)%trace_back_depth]
=max(STATE_METRIC[n1][ptr]+mac(msg[i][0],msg[i][1],-1*output[j][0],-1*output[j][1]),
STATE_METRIC[n1+1][ptr]+mac(msg[i][0],msg[i][1],output[j][0],output[j][1]));
}

ptr=(ptr+1)%trace_back_depth;

}

}

/////////////////// Find Maximum /////////////////////////////////////

int max(int a,int b)

{

        int maxi=a;

        if (b>a)

                maxi=b;

        return(maxi);

}

/////////////////// Traceback /////////////////////////////////////

void traceback(int d[],int end)

{

        short max_ind=0;

        for(int j=1;j<64;j++)

        {

                if(STATE_METRIC[j][ptr]>STATE_METRIC[max_ind][ptr])

                        max_ind=j;

        }
```

```
          for(int i=ptr-1;i>=end;i--)
           {
                      d[i]=max_ind/32;
                      max_ind=(max_ind & 31)*2;
                      if(STATE_METRIC[max_ind][i]<STATE_METRIC[max_ind+1][i])
                                 max_ind+=1;
           }
}
```

/////////////////////// MAC (Multiply Accumulate) //

int **mac**(int msg1,int msg2,int out1,int out2)

```
{
  return(msg1*out1+msg2*out2);
}
```

Vit_Data.h:

```
#ifndef vit_ data_h
#define vit_ data_h
void viterbi(int);
int max(int a,int b);
void traceback(int[ ], int);
int mac(int msg1,int msg2,int out1,int out2);
```

```
const int output[32][2] ={ {-15,-15},{15,-15},{15,15},{-15,15},{15,15},{-15,15},{-15,-15},{15,-15},{-15,-15},{15,-15},{15,15},{-15,15},{15,15},{-15,15},{-15,-15},{15,-15},

{-15,15},{15,15},{15,-15},{-15,-15},{15,-15},{-15,-15},{-15,15},{15,15},

{-15,15},{15,15},{15,-15}, {-15,-15},{15,-15},{-15,-15},{-15,15},{15,15} };

const short Branch_Metric[4][4]={{0,1,1,2},{1,0,2,1},{1,2,0,1},{2,1,1,0}};

#endif
```

A.3 ADSP-BF533 Assembly (Visual DSP++)

Trellis:

```
//////////////////////////////////////// TRELLIS FORMATION ////////////////////////////////////////////

#include <asm_sprt.h>

.section my_asm_section;

.global _trellis;

.extern _STATE_METRIC;

.extern _ptr;

.extern _output;

_trellis:

P2=R0;                  // i

P5=R1;

P0.h=_ptr;

P0.l=_ptr;

I0.h=_output;

I0.l=_output;

P3=10;          //siz

P2=P2+P5;   // msg[i][0]

P4=32;                  // no of states/2

I1.h=_STATE_METRIC;

R1=[P0];

R1=R1<<2;       // ptr

B3=0;

M1=R1; I3=R1;

I1.l=_STATE_METRIC;

I2=I1;

B2=I2;

M3=4;

B0=I0;
```

```
L0=128;

L3=404;

I1+=M1;

L2=25856;

I3+=M3;

M3=I3;

I2+=M3;

B1=I1;L1=25856;

M1=404;

M2=12928;

M3=12924;

M0=I2;

LSETUP (begin_loop, end_loop) LC0=P3;

begin_loop:

R0=[P2++];     // read msg[i][0]

R1=[P2++];     // read msg[i][1]

LSETUP (begin_loop2, end_loop2) LC1=P4;

begin_loop2:

R2.l=W[I0++];     // output[j][0]

R3.l=W[I0++]||A0=R0.l*R2.l(IS);

R4=(A0+=R1.l*R3.l)(IS)||R2=[I1++M1];

R6=[I1++M1]||R3=R2+R4,R7=R2-R4(S);

R2=R6+R4,R6=R6-R4(S);

R5=Max(R3,R6);

R7=max(R7,R2)||[I2++M2]=R5;

[I2++M2]=R7;

end_loop2:    I2+=M1;

I1=M0;

I2-=M3;

end_loop:     M0=I2;

RTS;

_trellis.end:
```

Trace back:

```
.section my_asm_section;

.global _traceback_asm;

.extern _STATE_METRIC;

_traceback_asm:


P2=400;

P0.l=_STATE_METRIC;

P0.h=_STATE_METRIC;

R6=P0;

P3=63;

P1=R0;

P0=P0+P2;

R0=[P0];

P2=404;

R3=1;

R2=31;

R4=0;

r1=[P0+404];

P0=P0+P2;

LSETUP (begin_loop, end_loop) LC0=P3;

begin_loop:

        cc=R1<=R0;

        if CC jump equ;

        R0=R1;

        R4=R3;


        equ:

        R3+=1;

        R1=[P0+404];

end_loop:    P0=P0+P2;


R7=101; // no of columns

P2=100;// Loop counter

R5=99;

M3=404;

L3=0;

LSETUP (begin_loop2, end_loop2) LC1=P2;

begin_loop2:
```

```
R3=R4>>5;

[P1++]=R3;

R4=R4&R2;

R4=R4<<1;

R7*=R4;

R3=R7+R5;

R3=R3<<2;

R3=R3+R6;

I3=R3;

R5+=-1;

R7=101;

R3=[I3++M3];

R0=[I3++M3];

CC=R3<R0;

if !CC jump end_loop2;

        R4+=1;

end_loop2: NOP;

RTS;

_traceback_asm.end:
```