Flexible constraint length Viterbi decoders on large wire-area interconnection topologies

A Thesis

Submitted For the Degree of

Master of Science (Engineering)

in the Faculty of Engineering

by

Ganesh Garga

Centre for Electronics Design and Technology

Indian Institute of Science

BANGALORE – 560 012

July 2009

To

My parents

Acknowledgements

I offer my sincere gratitude to both my research advisors - Prof. S. K. Nandy and Prof. H. S. Jamadagni for their constant guidance and patience during the three-year course of my degree. Prof. Nandy ably guided me to my research area, about which I had no idea when I entered the Institute. Subsequently, he participated in numerous discussions with me, and I have learned a lot from them. He also played a huge role in making my thesis readable. Prof. Jamadagni has also given his time and comments at certain important times during my degree. This thesis would not have assumed its current form had it not been for the sincere involvement and guidance provided by both my advisors.

A social support system is of utmost importance in order to make it through an academically demanding place like IISc successfully. I was lucky enough to know a large number of other students in IISc, and each of them has contributed to my mental well-being in some way or the other. I would like to thank all the members of IISc's music team, Rhythmica, of which I was a part, and which was like my family in IISc. The performances with Rhythmicans and other times spent in their company are some of the most joyous times I have experienced in life. I would however, specifically like to thank Janakiraman Viraraghavanraja and Deepika Vishwanathan for their very therapeutic presence in my life and for the most enjoyable tea, ice cream, lunch and dinner sessions. Many of these sessions were made even more joyous by the presence of Vinayak Raman, Tarandeep Singh, Arun Chandru, Avinash Acharya, Avinash Dawari, Vijay Venkatesh, Ganapathy S. E. K and Arulalan Rajan among many others. One of my department seniors, Kusum Lata, has also provided my with a great deal of guidance during my stay.

I was extremely fortunate to have a very dependable technical feedback group within

my lab itself - comprised of my labmates, Keshavan Varadarajan and Mythri Alle. They are almost as familiar with my work as I am myself. I have accosted them on innumerable occasions with some weird technical choice that is relevant only to me, and they have always patiently and whole-heartedly heard me out and provided their opinion. They have played a pivotal role in the shaping of my research. Over the past three years, they have also been my sound-board for airing my frustrations regarding various issues. My sustained sanity even after finishing this course is due, in a large part, to their patience in hearing my tantrums! I offer my sincere gratitude to them. Besides, I would also like to acknowledge the many enjoyable discussions, both technical and trivial, that I have had with my other labmates - Adarsha Rao, Amarnath Satrawala, Ritesh Rajore, Alexander Fell and Shrikant Joshi. I would like to specially thank Adarsha for having the courage to teach me motorbike-riding, using his own motorbike!

It is difficult to make a fully satisfying tribute to one's parents. My parents are my foundation. Any worthwhile action on my part, now and in the future, is just a fruit coming from the plant that they have cultivated over so many years, putting in all their sweat and blood. Let one such worthwhile action be my humble acknowledgement of their presence in my life and the boon that it has been for me all through.

Abstract

To achieve the goal of efficient "anytime, anywhere" communication, it is essential to develop mobile devices which can efficiently support *multiple* wireless communication standards. Also, in order to efficiently accommodate the further evolution of these standards, it should be possible to modify/upgrade the operation of the mobile devices without having to recall previously deployed devices. This is achievable if as much functionality of the mobile device as possible is provided through *software*. A mobile device which fits this description is called a *Software Defined Radio (SDR)*.

Reconfigurable hardware-based solutions are an attractive option for realizing SDRs as they can potentially provide a favourable combination of the flexibility of a DSP or a GPP *and* the efficiency of an ASIC. The work presented in this thesis discusses the development of efficient reconfigurable hardware for one of the most energy-intensive functionalities in the mobile device, namely, *Forward Error Correction (FEC)*. FEC is required in order to achieve reliable transfer of information at minimal transmit power levels. FEC is achieved by encoding the information in a process called *channel coding*. Previous studies have shown that the FEC unit accounts for around *40%* of the total energy consumption of the mobile unit. In addition, modern wireless standards also place the additional requirement of *flexibility* on the FEC unit. Thus, the FEC unit of the mobile device represents a considerable amount of computing ability that needs to be accommodated into a very small power, area and energy budget.

Two channel coding techniques have found widespread use in most modern wireless standards - namely *convolutional coding* and *turbo coding*. The *Viterbi* algorithm is most widely used for decoding convolutionally encoded sequences. It is possible to use

this algorithm *iteratively* in order to decode turbo codes. Hence, this thesis specifically focusses on developing architectures for flexible Viterbi decoders. Chapter 2 provides a description of the Viterbi and turbo decoding techniques.

The flexibility requirements placed on the Viterbi decoder by modern standards can be divided into two types - *code rate* flexibility and *constraint length* flexibility. The code rate dictates the *number of received bits* which are handled together as a *symbol* at the receiver. Hence, code rate flexibility needs to be built into the basic computing units which are used to implement the Viterbi algorithm. The constraint length dictates the number of computations required per received symbol as well as the manner of transfer of results between these computations. Hence, assuming that *multiple* processing units are used to perform the required computations, supporting constraint length flexibility necessitates changes in the *interconnection network* connecting the computing units. A constraint length K Viterbi decoder needs 2^{K-1} computations to be performed per received symbol. The results of the computations are exchanged among the computing units in order to prepare for the next received symbol. The communication pattern according to which these results are exchanged forms a *graph* called a *de Bruijn* graph, with 2^{K-1} nodes. This implies that providing constraint length flexibility requires being able to realize de Bruijn graphs of *various* sizes on the interconnection network connecting the processing units.

This thesis focusses on providing constraint length flexibility in an efficient manner. Quite clearly, the *topology* employed for interconnecting the processing units has a huge effect on the efficiency with which multiple constraint lengths can be supported. This thesis aims to explore the usefulness of interconnection topologies *similar* to the de Bruijn graph, for building constraint length flexible Viterbi decoders. Five different topologies have been considered in this thesis, which can be discussed under two different headings, as done below:

De Bruijn network-based architectures

The interconnection network that is of chief interest in this thesis is the de Bruijn interconnection network itself, as it is identical to the communication pattern for a Viterbi decoder of a given constraint length. The problem of realizing flexible constraint length Viterbi decoders using a de Bruijn network has been approached in two different ways. The first is an embedding-theoretic approach where the problem of supporting multiple constraint lengths on a de Bruijn network is seen as a problem of *embedding* smaller sized de Bruijn graphs on a larger de Bruijn graph. Mathematical manipulations are presented to show that this embedding can generally be accomplished with a maximum dilation of $\frac{log(N)}{2}$, where N is the number of computing nodes in the physical network, while simultaneously avoiding any congestion of the physical links. In this case, however, the mapping of the decoder states onto the processing nodes is assumed fixed.

Another scheme is derived based on a *variable* assignment of decoder states onto computing nodes, which turns out to be more efficient than the embedding-based approach. For this scheme, the maximum number of cycles per stage is found to be limited to 2 *irrespective* of the maximum contraint length to be supported. In addition, it is also found to be possible to execute *multiple* smaller decoders in parallel on the physical network, for smaller constraint lengths. Consequently, post logic-synthesis, this architecture is found to be more area-efficient than the architecture based on the embedding theoretic approach. It is also a more efficiently scalable architecture.

Alternative architectures

There are several interconnection topologies which are closely connected to the de Bruijn graph, and hence could form attractive alternatives for realizing flexbile constraint length Viterbi decoders. We consider two more topologies from this class - namely, the *shuffle-exchange* network and the *flattened butterfly* network. The variable state assignment scheme developed for the de Bruijn network is found to be directly applicable to the shuffle-exchange network. The average number of clock cycles per stage is found to

be limited to 4 in this case. This is again independent of the constraint length to be supported. On the flattened butterfly (which is actually identical to the *hypercube*), a state scheduling scheme similar to that of *bitonic sorting* is used. This architecture is found to offer the ideal throughput of one decoded bit *every* clock cycle, for any constraint length.

For comparison with a more general purpose topology, we consider a flexible constraint length Viterbi decoder architecture based on a *2D-mesh*, which is a popular choice for general purpose applications, as well as many signal processing applications. The state scheduling scheme used here is also similar to that used for bitonic sorting on a mesh.

All the alternative architectures are capable of executing multiple smaller decoders in parallel on the larger interconnection network.

Inferences

Following logic synthesis and power estimation, it is found that the de Bruijn network-based architecture with the variable state assignment scheme yields the lowest $(area) - (time)$ product, while the flattened butterfly network-based architecture yields the lowest $(area) - (time)^2$ product. This means, that the de Bruijn network-based architecture is the best choice for moderate throughput applications, while the flattened butterfly network-based architecture is the best choice for high throughput applications. However, as the flattened butterfly network is less scalable in terms of size compared to the de Bruijn network, it can be concluded that among the architectures considered in this thesis, the de Bruijn network-based architecture with the variable state assignment scheme is overall an attractive choice for realizing flexible constraint length Viterbi decoders.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This chapter briefly describes the evolution of mobile communication technology through the past 4 decades. Further, it places the work presented in this thesis in context, through a description of the various functions involved in a typical modern wireless communication system. Finally, a high-level description of the requirements placed on the *channel decoder* by modern (3G and above) wireless standards is provided. Efficient, flexible and area-optimal *multiprocessor* realization of channel decoding algorithms is the focus of this thesis. The chapter concludes with an outline of the work presented in this thesis followed by a description of the organization of the succeeding chapters of the thesis.

1.1 Evolution of mobile communication technology

Mobile communication has transformed the way the world and society functions at large in the last 40 years. Mobile communication technology has evolved through several *generations* in these past years, providing more extensive and robust features with each subsequent generation. After Motorola, in conjunction with Bell telephone company, operated the first commercial mobile telephone service, called *MTS* in the US in 1946, a number of other commercial services were started in various countries (*Televerket* in Norway and *B-Netz* in West Germany among many others). These constitute the earliest

generation (called *0G*) of mobile communications. *Analog* signalling (where the voice signal is directly transmitted, after upconversion to a higher frequency) was used in all these communication systems. The services graduated from being manually operated at first, to being able to set up calls automatically between mobiles. These systems only provided the basic voice transmission.

Subsequently, mobile communication evolved to so-called *1G* in the 1980s, where the technology became more standardized and mature. Examples of 1G standards are the *Nordic Mobile Telephone (NMT)* introduced in the Nordic countries, *Advanced Mobile Telephone System (AMTS)* introduced in the United States and Australia and *Total Access Communication System (TACS)* introduced in the United Kingdom. In these standards, *digital* signalling was used for communication between radio towers (each of which communicate with mobile phones in a particular area) and analog signalling was used for communication between individual mobile phones and the radio tower. The service provided still comprised only voice communication.

Analog signalling was removed from mobile communication in the subsequent generation of networks, the *2G* generation, in the 1990s. Examples of 2G standards, which are also in use today, are the *Global System for Mobile Communications (GSM)* which was launched first in Finland in 1991 and the *Interim Standard-95 (IS-95)* or *cdmaOne* standard, which was launched in the United States (where it is commonly called CDMA). The main difference between GSM and IS-95 is in the method of *multiplexing* multiple voice or signalling channels for transmission purposes. While *Time Division Multiplexing (TDM)* is used for multiplexing several speech or signalling channels in GSM, *Code Division Multiple Access (CDMA)* is used for the same purpose in IS-95. There were three primary advancements in this generation over the previous generation:

1. Phone conversations were digitally *encrypted*.

2. The available communication channel capacity was used much more efficiently than in 1G, which allowed for greater pervasion of the technology.

3. *Data* services were introduced for the first time, starting with the *Short Message*

Service (SMS).

Some enhancements were made subsequently in these standards in order to obtain greater efficiency. *Packet-switching* was employed for network access in addition to the traditional circuit-switching technique. In packet-switching, a message is broken into several blocks (called "packets") and successive packets are routed in *different* ways from the source to the destination. This allows for greater routing flexibility compared to the circuit-switching network access technique, wherein a particular route through the network is *bound* to a given source-destination pair and all messages are transferred from the source to the destination only along this route. The standards incorporating the packet-switched network access technique are grouped into *2.5G* or *2.75G* standards. Examples are the *Generalized Packet Radio Service (GPRS)* and *Enhanced Data Rates for GSM Evolution (EDGE)* for GSM. The *Multimedia Messaging Service (MMS)* was also introduced in these standards.

Subsequently, the most recent technology generation in commercial use, called *3G* was introduced. It was launched first by NTT DoCoMo in Japan in 2001, based on *Wideband-Code Division Multiple Access (W-CDMA)* technology which is the 3G successor of the 2G GSM technology. In the US, the 2G IS-95 standard has evolved to the *CDMA2000* 3G standard, operated by Verizon Wireless since 2003. 3G provides a range of additional services over 2G communication, including *video calls*, *broadband wireless data transfer* and high speed data transmission of the order of tens of megabits per second, which is delivered by standard extensions like *High Speed Packet Access (HSPA)*, all in a mobile environment. Technology has also evolved for short-range high-bandwidth networks primarily for data communication, for instance the *IEEE 802.11* set of standards, which includes *Wi-Fi* and *WLAN* (around 50 Mbps). In December 2007, 190 3G networks were operating in 40 countries, according to the Global Mobile Suppliers Association. Efforts are on to develop a common world-wide standard for 3G communication, mainly through the *3rd Generation Partnership Project (3GPP* and *3GPP2).*

The demand for mobile communication is great, as can be inferred from the fact that by June 2007, the 200 millionth 3G subscriber had been connected. The next complete

evolution in mobile communication, expected to arrive in 2012-2015, is referred to as *4G*. A 4G system is conceived to be a fully *IP-based system*, capable of providing voice, data and streamed multimedia services on an "anytime, anywhere" (in other words, plug-and-play) basis, and at higher data rates than previous generations (in the *hundreds* of megabits and even *giga*-bits range), in addition to customized quality-of-service (QoS).

An important objective to be achieved in 4G is *efficient utilization* of the available spectrum, in order to be able to accommodate as many users as possible. To achieve this, the trend in both 3G and 4G standards is to incorporate a considerable amount of *flexibility* in the manner in which mobile devices and base-stations/central towers communicate. Even better spectrum utilization can be obtained by building wireless communication devices which can support *multiple* communication standards, wherein each standard is itself considerably flexible and is geared to efficiently support a particular type of transmission. Such devices are also expected to have the intelligence to decide on the *best* standard for a certain type of communication. For example, if a mobile device has access to a WCDMA network as well as a Wi-Fi network in a certain area, the device should be capable of choosing the Wi-Fi network when it wants to transfer *non-voice data*(as the Wi-Fi network is geared to more efficiently support high data transfer rates), and the WCDMA network when *voice* communication is needed. It is clear that providing the kind of flexibility outlined above is easiest if the mobile device functionality is implemented entirely in *software*. A radio based on this concept is called *software radio*, a term coined by Joseph Mitola in 1991. A software radio requires the signal received from the antenna to be directly *digitized*, and all subsequent stages (which operate on digital data) to be implemented in software. However, due to limitations of contemporary analog/digital conversion technology, the ideal concept of software radio has been scaled down to a more practical concept, called *software defined radio*. A software defined radio is the sum of a software radio and a set of *analog* circuits, wherein the analog circuits transform the signal received at the antenna into a form which can be fed to an analog-to-digital convertor (ADC) and subsequent stages are developed using digital technology. The flexibility is typically built partly *into* the digital hardware, and

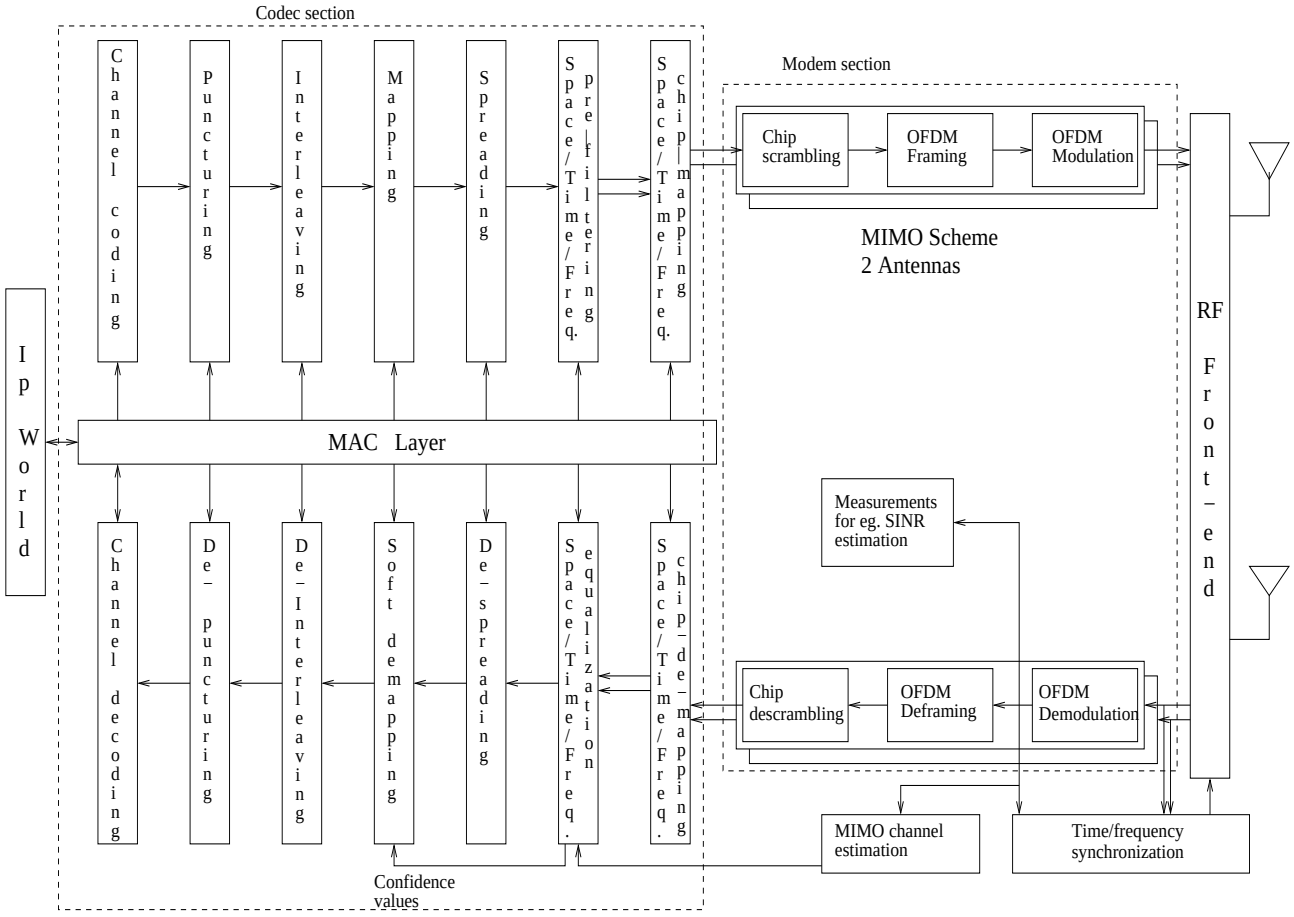


Figure 1.1: 4More wireless receiver functional diagram.

partly into the *software* running on the digital hardware.

1.2 Basic functions in a typical wireless communication system

The functions involved in a typical wireless communication system are shown in the block diagram of figure 1.1. The specific standard shown is the very recent 4More air interface defined by the IST project of the same name [26]. Both the transmission and reception processes have been shown in figure 1.1.

All modern communication standards use digital communication. This means that even voice data is converted to digital form (through an *analog-to-digital converter*

(ADC), before transmission. Ultimately, the information to be transmitted is always in digital form.

At the transmitter, *five* major modifications are made to this digital information:

1. The digital *information bit* stream is first *encoded*. Encoding broadly refers to the careful addition of *redundancy* to the information bit stream in order to increase the probability of reliable reception of the information sequence even in the presence of noise. A process called *puncturing* controls the amount of redundancy added to the information bit stream.

2. Secondly, the encoded bit stream is *spread*. This refers to the process of multiplication of the encoded bit stream with a specific bit pattern, in order to implement the CDMA functionality. Multiple information streams are multiplied with *mutually orthogonal* bit patterns, so that it is possible to transmit and receive all the streams over the same communication channel without the streams interfering with each other. The bit pattern used for multiplication (spreading sequence) has a much higher bit rate than the encoded bit stream. This causes a single encoded bit to "spread" over multiple spreading sequence bits, hence the name "spreading".

3. Thirdly, the encoded and spread digital sequence is *scrambled*. Scrambling is the process of making the communication *secure*, so that an unwanted listener is not able to obtain the transferred information. It involves the multiplication of the encoded digital sequence with a much higher rate *pseudo-noise* digital sequence.

4. Subsequently, the scrambled and encoded signal is converted to an equivalent signal in the *analog* domain, by the process of *modulation*. In this process, one of the parameters of the analog signal, which may be the amplitude or the phase or a combination of the two, is made dependent on the polarity of the bit to be transmitted.

5. Lastly, the modulated analog signal is transferred to a much higher frequency for transmission through the process of *upconversion*. This is achieved through the use

of an analog *mixer* and a local oscillator. This section also involves some amount of *filtering*, in order to remove unwanted noise and to control the *bandwidth* occupied by the transmitted signal.

At the receiver, the *complements* of the above functions are performed in *reverse* order in order to reobtain the original digital information. This includes *downconversion*, *demodulation*, *descrambling*, *despreading*, *depuncturing* and *decoding* in that order. Besides these, *synchronization* circuits are needed in the receiver in order to correctly detect the transitions of the demodulated signal for further processing.

With reference to figure 1.1, upconversion/downconversion is performed in the *RF front-end* block of figure 1.1. Among the other functions, (de)modulation and (de)scrambling are combined in the *modem* section, while (de)spreading, (de)puncturing and (de)coding are combined in the *codec* section.

As mentioned before, the work in this thesis relates to the channel decoder, which is part of the *codec* section of the wireless communication receiver.

1.3 Flexibility requirements placed on channel decoders in modern wireless standards

In modern (3G and above) wireless standards, efforts are made to match the capabilities of the receiver to the conditions in the communication channel. For instance, if the channel has a significantly low noise level, a relatively less robust error correction scheme would serve the purpose. If the channel has a high noise level, a more robust error correction scheme would be necessary. Hence, modern channel decoders are expected to provide support for *several* channel decoding algorithms, with the ability to *switch* from one algorithm to another dynamically during the course of a conversation. Furthermore, for each supported channel decoding algorithm, various *algorithm-level* parameters are also expected to be *flexible*, in order to finely match the instantaneous communication channel conditions. Moreover, with the evolution of wireless standards, the expected *data*

rate has also been steadily increasing. Due to the two-fold expectation of increased flexibility as well as increased throughput, the design of modern channel decoders presents a major challenge, which warrants a thorough evaluation of various existing as well as *new* design approaches in order to identify the best possible approach.

1.4 Thesis outline and contributions

The work in this thesis is directed towards specifically providing *constraint length* flexibility in *Viterbi* decoders. The motivation for this work is provided in chapter 2. There are two major contributions of the work in this thesis:

1. *Two* approaches for building flexible constraint length Viterbi decoders on a *de Bruijn* interconnection network of processing elements. The first approach views the problem as a *graph embedding* problem, while the second approach takes a more hardware-centric view.

2. Comparison of the silicon area-efficiency of the de Bruijn-based Viterbi decoder architecture with three other proposed architectures based on the *shuffle-exchange* network, the *flattened butterfly* network and the *2D-mesh* network respectively.

The succeeding chapters in the thesis are arranged as follows. Chapter 2 the two most commonly used channel decoding algorithms, namely *convolutional* coding and *turbo* coding in detail, and interprets the requirements of modern communication standards in their context. It also provides a survey of previous literature related to the design of Viterbi/turbo decoders. Chapter 3 first provides a brief survey of some recent approaches taken towards the design of *modern/flexible* channel decoders and motivates the approach taken in this thesis. It then describes the two de Bruijn interconnection network-based architectures in detail. Chapter 4 describes the other three architectures using the shuffle-exchange, flattened butterfly and the 2D-mesh interconnection networks. It also provides the synthesis results of all the architectures considered and draws inferences about the area-efficiency of the architectures. Chapter 6 finally draws conclusions from the work presented in the thesis and outlines some directions for future work.

Chapter 2

Channel decoding algorithms

In this chapter, a detailed description of the two most commonly used channel coding techniques - namely, *convolutional* coding and *turbo* coding is presented. Further, a survey of *prior* literature related to various issues in the implementation of these decoders is also presented. Finally, the specifications of the channel decoder as defined in modern standards is interpreted in the context of Viterbi/turbo decoders.

2.1 Overview of channel coding and decoding

Channel coding/forward error correction (FEC) is a technique used to minimize the transmit power level needed in order to achieve an acceptably reliable transfer of information between two communicating mobile units. The basic idea is to add some *redundant* bits to the actual information stream in order to be able to *reconstruct* the information bits at the receiver, even if some of the bits are received erroneously (due to the lower transmit power level). The process of adding the redundant bits to the bit stream at the transmitter is called *encoding* and the process of obtaining the actual information bits from the received bit stream at the receiver is called *decoding*.

2.2 Convolutional encoding

The convolutional encoder consists of a shift register, and a set of XOR gates, where each gate accepts some of the bits of the shift register as inputs (refer figure 2.1). The shift register is initialized to 0 at the start of the encoding process. For each input information bit, the outputs of these gates are given out as the encoded bits. Also, the contents of the shift register are shifted once in the direction of the incoming input stream, and the input information bit is shifted into the register. This process is repeated for all input information bits.

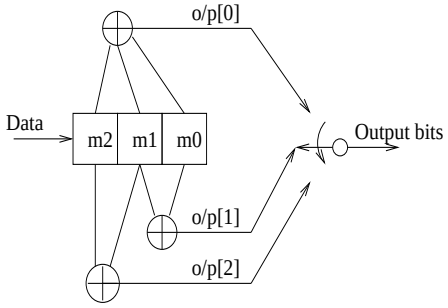


Figure 2.1: Example convolutional encoder. The input is fed into the leftmost shift register bit. At each information bit interval, the contents of the shift register are shifted once to the right, and the outputs of the three XOR gates are serially given out as the encoder outputs.

Two parameters related to this encoder are of importance. If k output bits are produced for n input bits ($k < n$), then the encoder is said to have a *code rate* of $\frac{k}{n}$. For the above encoder, the code rate is 1/3. As a particular information bit passes through the shift register, it influences the generation of *multiple* sets of encoded bits. Each set is called a *symbol*. The number of bit positions in which a particular input bit *can* influence the generation of the encoded bits is called the *constraint length*. For the above encoder, the constraint length is 4 - consisting of the three shift register bit positions and the input bit position itself. (However, in this particular encoder, the input bit is not directly XORed to obtain any output bit).

Thus, the receiver receives a *sequence* of symbols at its antenna.

The specific shift register bits which are XORed to form a particular output bit are

specified through a *generator polynomial*. This polynomial is formed as follows:

1. Assign a power of some variable (say x) to each bit position in the encoder shift register. For instance, in the shift register of figure 2.1, let us assume that the rightmost bit is assigned $x^0 = 1$, the bit to its left is assigned x^1 and so on.

2. The generator polynomial for a particular output bit is expressed as the sum of all powers associated with those shift register bits which are XORed to produce the output bit. For instance, in figure 2.1, the generator polynomial associated with $o/p[0]$ is $x^2 + x^1 + 1$. The generator polynomial can also be expressed as the ordered set of numbers [1,1,1], where a 1 is written for each bit position which is fed into the XOR gate for producing a particular output, and a 0 is written for each bit position which is not used for producing the output. The numbers are ordered according to the decreasing order of the power associated with the bits of the encoder shift register (ie. the first number corresponds to x^2, the second number corresponds to x^1 and the third number corresponds to x^0).

The encoder can be viewed as a Finite State Machine (FSM), where the contents of the shift register define the *state* of the machine at any given time. At each time instant, in accordance to the input information bit, certain output bits are produced and the encoder goes to a new state due to the shifting of the register contents. As such, the operation of the encoder can be captured by the state transition diagram shown in figure 2.2:

The state is defined by the three stored bits in the shift register (the leftmost bit is taken as the MSB). As each information bit is streamed in, the present state in these registers transitions to a new state, as depicted in the state transition diagram, where the transitions occurring upon a '0' input bit are shown as normal arrows while the transitions occurring upon a '1' input bit are shown as dotted arrows. With each transition, certain output bits are generated. The arrows in the state diagram have been annotated with the corresponding input bits and generated output bits (the output bits have been arranged as (o/p[0],o/p[1],o/p[2])).

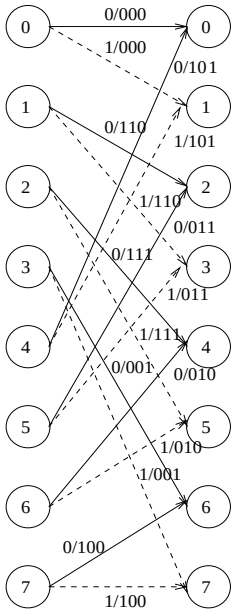


Figure 2.2: State transition diagram for the encoder in figure 2.1

2.3 Convolutional decoding

Viterbi decoders are used to decode convolutionally encoded information sequences. The basic algorithm [81] was proposed by Andrew Viterbi in 1967 for decoding convolutionally encoded data symbols. In 1969, J. Omura [66] showed that the Viterbi algorithm was a dynamic programming solution to the problem of finding the shortest path through a weighted graph, where the graph in case of convolutional codes refers to the *trellis diagram* (explained later). In 1973, Forney [37] showed that the Viterbi algorithm can be used to find the *most likely* transmitted information bit sequence from a convolutionally encoded symbol stream sent over a noisy channel.

For the purpose of explaining the algorithm, it is useful to refer to the *trellis* diagram, shown in figure 2.3.

The trellis diagram is just the encoder state transition diagram repeated as many times as the number of information bits. At the receiver, encoded symbols corresponding to these information bits are received, from which the corresponding information bits have to be derived. Now, at the transmitter end, the input bits cause a particular *set* of *state transitions* to occur in the encoder, which corresponds to a *unique* path through

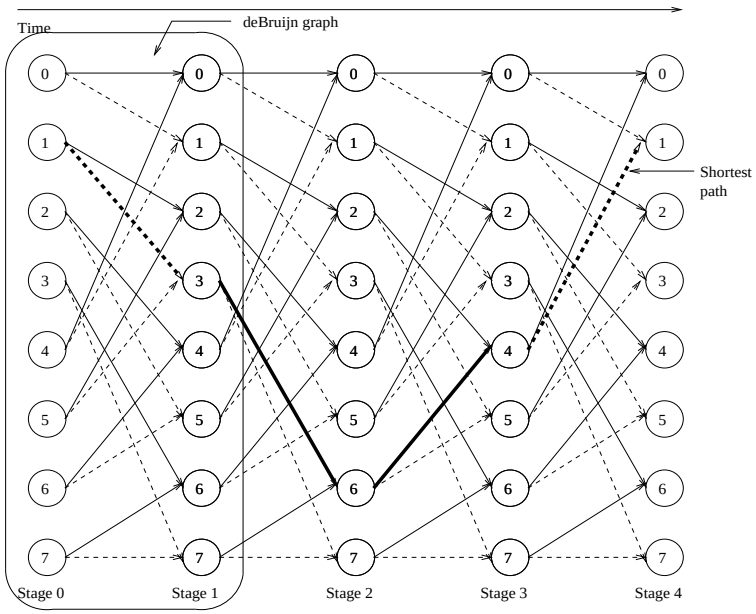


Figure 2.3: Trellis diagram for the state transition diagram shown in figure 2.2

the trellis diagram (One such path is shown in bold in figure 2.3). At the receiver, if this path can be identified among the many possible paths through the trellis (based on the received encoded bits), then the corresponding information bits can be regenerated. The Viterbi algorithm estimates the *most likely* path followed at the transmitter end, thus enabling the reconstruction of the most likely transmitted information *bit sequence*. This is different from the most likely transmitted *bit* estimation, which is performed by algorithms like MAP and log MAP [71]. The way the Viterbi algorithm performs this estimation is by computing for a given input symbol sequence, a *metric* for every possible path through the trellis diagram, which characterizes the level of dissimilarity between the received symbols and the symbols that would have been transmitted, had the concerned path been followed at the transmitter end. This metric (called the *path metric* or *accumulated metric*) is updated one input symbol at a time. At the end of the input symbol sequence, the algorithm chooses the path with the *least* accumulated metric as the most likely followed path, from where the most likely transmitted information bit sequence can be obtained. The metric can be obtained in various ways. The most elementary method is to calculate the *Hamming* distance between the output symbol that would have been generated by a particular state transition and the symbol received

for that time interval from the channel. The Hamming distance is the total number of bit positions in which two binary numbers are different, in terms of polarity. This quantity is accumulated for each possible path through the trellis across all time intervals/received symbols.

Note that there is a potential problem of exponentially growing computational load per stage in the method stated above. This can be brought out as follows. Consider state 0 at stage 2 of the trellis in figure 2.3. There are two incoming transitions into this state, and two outgoing transitions from this state. Thus, there are *four* possible paths through this particular state. At the next stage (stage 3), *two* such sets of four possible paths have to be considered for metric calculation at state 0 (one coming from state 0 of stage 2, and another coming from state 4 of stage 2).

To avoid this problem, one of the two incoming paths at each state of each stage is eliminated, and only one is retained. This causes the number of paths to be considered at each time interval to remain constant. This elimination is possible due to a simple property of the accumulated metric: it is a monotonically rising quantity. Hence, of the two incoming paths at a state within a stage, the one with a higher accumulated metric will continue to remain the less likely path, irrespective of the received symbols considered after that stage. Hence, this path can be safely eliminated from future consideration and only the *surviving* paths through each state considered from the next stage onwards.

To perform this selection of the surviving path through a particular state of a particular stage, an *Add-Compare-Select (ACS)* operation is performed. In the Add stage, the *incremental* metrics (also called the *branch* metrics) for the two incoming paths corresponding to the *most recent* state transitions (in other words, corresponding to the current received symbol) are added to the corresponding accumulated metrics at the *previous* stage on the two incoming paths. The following example will better illustrate this operation. Suppose that at some state s, there are two incoming paths, one with an accumulated metric of 5 and another with an accumulated metric of 6, taking into consideration all input symbols until time $(t-1)$, where t is the current time instant. Assume that the decoder is a rate $\frac{1}{2}$ decoder, and the received symbol at time t

is 10. Also, assume that the latest transition on the first path (corresponding to time interval t) passing through state s is associated with the output bits 11 and the latest transition on the second path passing through state s is associated with the output bits 01. If the Hamming metric is used, then the branch metric for the first path will be $(1 \oplus 1 + 0 \oplus 1) = (0 + 1) = 1$ and the branch metric for the second path will be $(1 \oplus 0 + 0 \oplus 1) = (1 + 1) = 2$. The updated path metrics would be $5 + 1 = 6$ for the first path and $6 + 2 = 8$.

In the Compare-Select stage, the smaller of the accumulated metrics is identified and the corresponding path (the first path in the previous example) is selected as the survivor path while the other path is eliminated. The smaller accumulated metric is *stored* in memory for retrieval during the next stage of ACS computation. The decision made is also stored for future use in identifying the most likely path.

Note that *dual* storage for the path metrics is generally needed as updated metrics to be consumed at the next stage of computation may arrive before the present values of the same metrics are consumed in the current stage. This is typically implemented through the use of a *ping-pong* memory. A ping-pong memory consists of two memory bank. The ACS units write the updated path metrics into one of the banks while reading from the other in one stage computation. In the next stage computation, the roles of the memory banks are interchanged.

At the end of the incoming symbol sequence, there are a set of survivor paths, one passing through each state, out of which the most likely followed path has to be identified, in accordance with the algorithm. One way is to find the path with the least accumulated metric among the paths passing through each of the states by comparing all the accumulated metrics. This technique becomes difficult to realize when the number of states (paths) becomes large, as is the case in most modern standards where a constraint length of 9 (which corresponds to an encoder with 256 states) is common. Another way of achieving this is to stream in special *tail* bits into the encoder, following the last information bit, which cause the encoder to end up in a predetermined state after the transmission. This last state, which is obviously on the most likely path, can be

precommunicated to the receiver, so that the most likely path can be uniquely identified at the receiver from among the survivor paths. For both these methods, however, it is necessary to remember the *entire* path for each of the states in the encoder. In practice, the length of the transmitted sequence is large enough that it is not practical for the receiver to remember the entire path. Fortunately, in Viterbi decoders, the survivor paths display a *merging* property. Suppose that the algorithm has proceeded for some $(m + L)$ stages. It is found that all surviving paths at stage $(m + L)$ pass through the *same* state (say s) at stage m. This means that state s is definitely on the final survivor path. Consequently, the survivor path can be traced backwards from state s and the decoded bits given out as the output sequence (how to obtain the decoded bits from the survivor path is explained in the next paragraph). Due to this merging property, it is only necessary to retain survivor path history which is L stages long. In practice, this L (called the *truncation length*) has been fixed through various simulations at roughly 5 times the constraint length (refer [56, 67] for more explanation on the causes for this behaviour). This allows us to decode arbitrarily long information sequences, by only remembering path lengths upto about 5 times the constraint length.

There are two basic methods by which these paths can be retraced and the decoded bits obtained - the *traceback* method and the *register exchange* method. In the traceback method, for each state in each stage, the *decision* made at that stage is stored. After an appropriate number of stages (at least equal to the truncation length) have elapsed, a separate process is started in order to retrace the survivor path based on the stored decisions. This retracing is done in an iterative manner. Starting with a state on the survivor path at the most recently computed stage (say S_t), the state on the same path at the *previous* stage (S_{t-1}) is determined using the stored decision at S_t for that state (how this is done is explained in more detail together with other path retracing methods in section 2.5). The process is repeated for all previous states in order to *trace back* the survivor path $(m+L)$ stages back, where the survivor memory is $(m+L)$ stages long. As the previous state is computed from a given current state, it is also possible to determine that information bit which would have caused that particular transition to occur. After

L stages of traceback, the information bits causing the next m current state-to-previous state transitions can be delivered as decoded output bits.

In the register exchange method, the *entire* path upto the current stage is stored, in terms of information bits causing the various transitions, for each state. Once the survivor path through a state at a given stage (S_t) is determined through the ACS operation, the *entire* path information associated with the state on the survivor path at the previous stage (S_{t-1}) is accepted, the current transition is appended to it, and the new survivor path information is stored at the location associated with the state under consideration. Since the entire path information is available at each state, there is no need for a separate process to obtain the decoded bits. Obtaining the decoded bits is reduced to reading the appropriate bits off a register.

In the traceback method, a decision bit is stored for each state of the decoder for a number of stages which is at least equal to the truncation depth. Thus, ($L\times$ *(no. of states)*) bits are required to be stored in all. In the register exchange method, a word comprised of decision bits which is at least L bits long is required to be stored for each state, which also leads to a total storage requirement of ($L\times$ *(no. of states)*). Thus, the total amount of memory needed for survivor path storage is the *same* in both the methods. However, in the register exchange method, a word containing the entire path information at a state S_{t-1} at stage $(t-1)$ is transferred to a state S_t at stage t. If a constraint length of 9 is considered, this word needs to be at least $9 \times 5 = 45$ bits long. Thus, the register exchange method has a considerably larger *interconnection* requirement. However, it eliminates the need to explicitly trace back paths, due to which higher throughputs can be obtained. These as well as other recent methods for obtaining the output bits are explained in detail in chapter 2.5.

2.4 Turbo encoding and decoding

An example turbo encoder is shown in figure 2.4.

The encoder consists of two *Recursive Systematic Convolutional (RSC)* encoders.

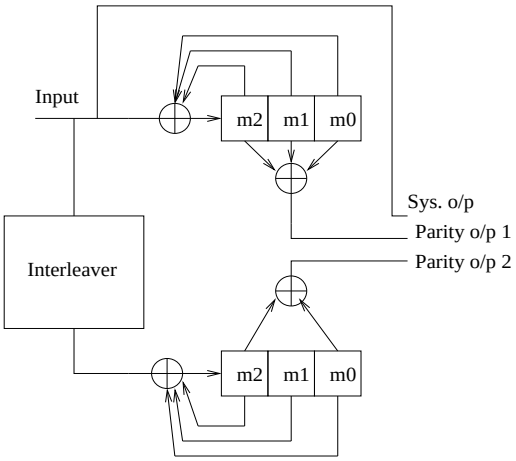


Figure 2.4: Example turbo encoder

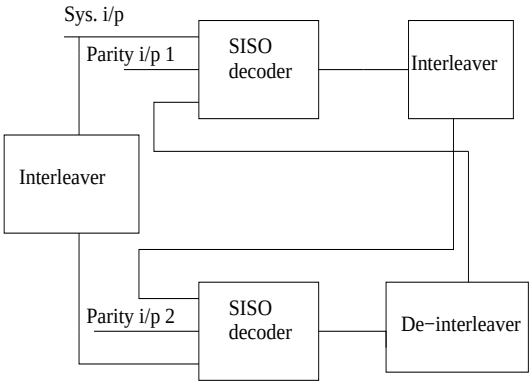


Figure 2.5: The turbo decoder

These are shift-register based encoders similar to convolutional encoders, but in which, the bit shifted in at each step is determined by the input bit *as well as* the current state of the encoder. One such encoder works on the input data bits in their original order, while the other works on the input data in a *permuted* order. The permutation is realized through an interleaver. Each bit is transmitted from the encoder along with the two associated parity bits, making this encoder a $\frac{1}{3}$ encoder. As the RSC encoders have a shift register length of 3, the constraint length of the turbo encoder is 4.

The corresponding turbo *decoder* is shown in figure 2.5.

The turbo decoder comprises of *two* Soft-Input-Soft-Output (SISO) decoders which are connected in a feedback loop. Each decoder accepts *soft* or *quantized* inputs and generates *soft outputs* (where each decoded bit is annotated with a *measure* of certainty

about its value). Two key algorithms have been proposed for SISO decoding - the Soft-Output-Viterbi-Algorithm (SOVA) [42] and the Maximum A-posteriori Probability (MAP) algorithm [71]. Both these algorithms are based on traversal of the trellis diagram similar to Viterbi decoding.

The upper decoder works on the received versions of the original input bit and one of the parity bits, while the lower decoder works on the *permuted* version of the received original bit and the other parity bit. The upper decoder obtains estimates of the information bits based on its inputs, and feeds this information to the lower decoder. The lower decoder uses both its own inputs *and* the estimates provided by the upper decoder to estimate the information bits. These estimates are fed back to the upper decoder, which proceeds to the next iteration and generates new estimates of the information bits. A number of such iterations (typically 4) are performed, with the Bit Error Rate (BER) reducing with each successive iteration. The final information bit estimates after all the iterations are over are given out as outputs after performing the necessary quantization of the *soft* values into *hard* (0 or 1) bit polarities.

As the SISO algorithms are also based on the traversal of the trellis, the computation and communication requirements of the SISO decoders are analogous to Viterbi decoders. This indicates that efficient architectures for Viterbi decoding can be naturally extended to accommodate turbo decoding as well. Hence, in this thesis, we focus on efficient architectures for Viterbi decoding as a conceptual starting point for research into more general channel decoder architectures.

Previous research in Viterbi decoders, or more generally, in channel decoders, has focussed on various diverse topics. It can be suitably represented as shown in figure 2.6.

The next section provides a survey of previous results regarding various issues in the implementation of conventional as well as modern/flexible Viterbi/turbo decoders. The aim behind including this survey is to provide more insight into the properties of the Viterbi/turbo decoder, and also to bring together various distributed and well-known innovations which are today a part of the standard optimization process in any decoder design.

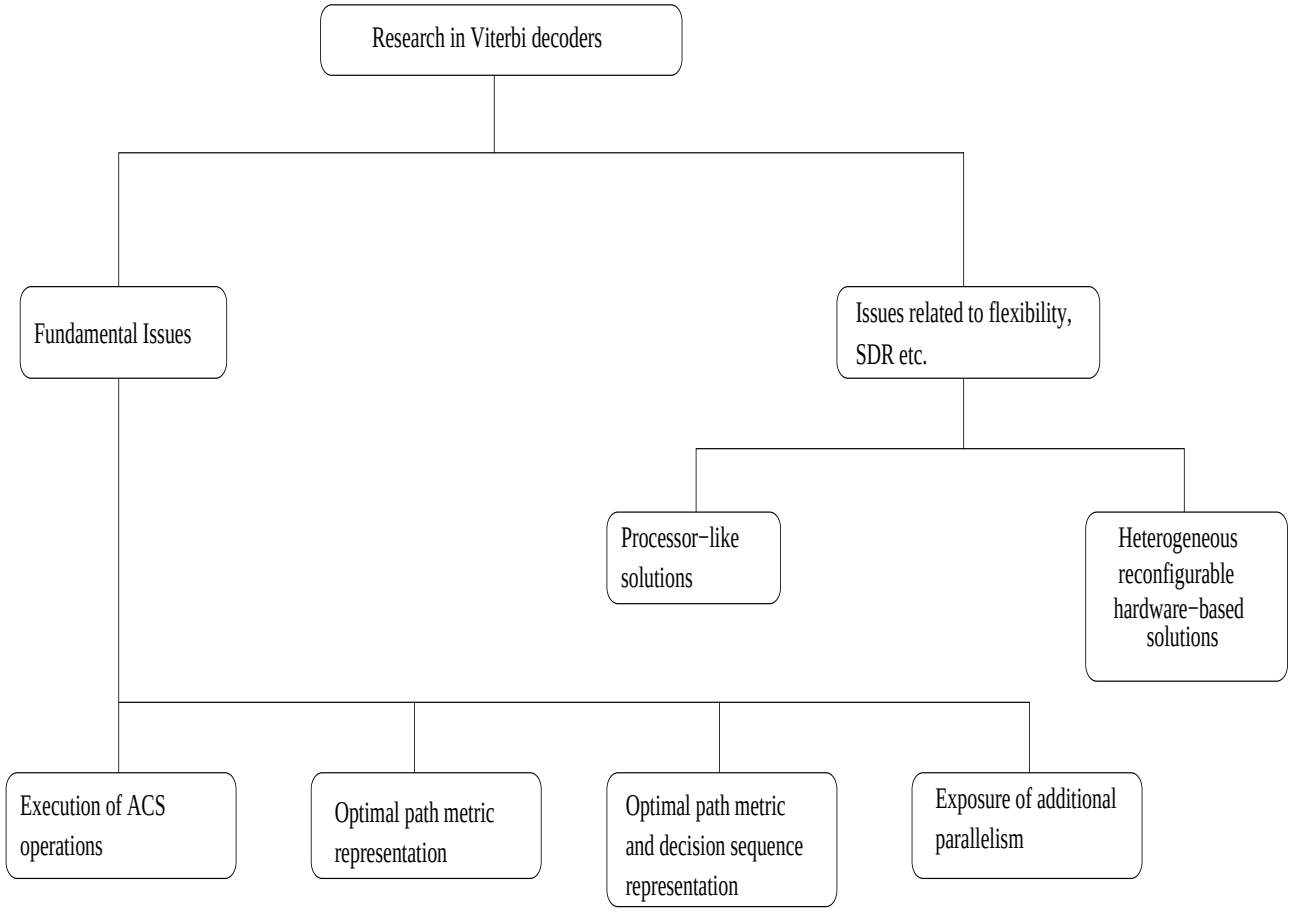


Figure 2.6: Broad subdivision of previous research efforts in the area of Viterbi decoders

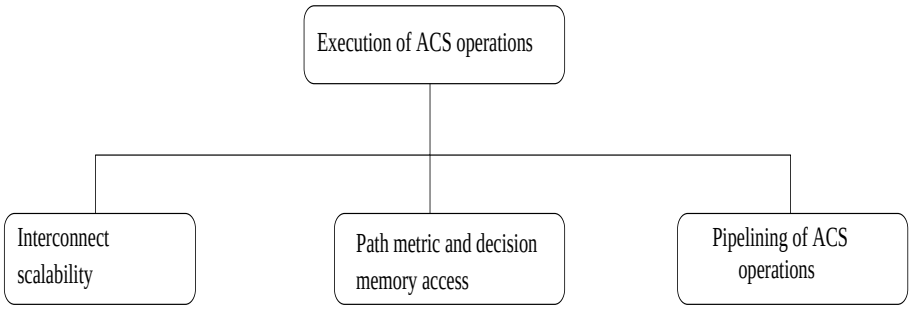


Figure 2.7: Subdivision of research related to execution of ACS operations in Viterbi decoders.

2.5 Previous results related to Viterbi decoder implementation

The specific fundamental issues on which prior research has focused can be dividied into four topics as given below:

1. Execution of the basic ACS operation and efficient transport of path metrics.
2. Optimal bit representation of the path metrics.
3. Optimal survivor path representation and management.
4. Methods of extracting more parallelism in the algorithm in order to obtain higher throughputs and/or more efficient use of the hardware resources.

2.5.1 Execution of the basic ACS operation and efficient transport of path metrics

The ACS computations can be performed on the array of ACS units in two ways as listed below:

1. State-sequential or intermediately state-parallel (a few ACS units on which the ACS computations for all states in each stage are performed in a sequential manner).
2. State-parallel (one ACS unit for each state in the trellis, on which each stage is executed in one clock cycle).

State-sequential approach

Using only *one* ACS unit on which the ACS computations of all states within a stage are executed leads to the minimum possible consumption of silicon area. However, the throughput is also adversely affected. Such a solution is not capable of providing the throughputs required in modern standards and hence a completely state-sequential approach has not received significant attention in the literature. A lot of attention has been given to architectures with an *intermediate* amount of parallelism, where P ACS units are used to execute N ACS operations per stage in a sequential manner ($N > P$). Each ACS unit executes $\lceil \frac{N}{P} \rceil$ ACS operations at every stage. Many previous efforts present architectures in this category and provide different kinds of tradeoffs between area and throughput.

Architectural frameworks: *Architecture scalability* is a major issue in the design of modern channel decoders. The range of constraint lengths, code rates and generator polynomials required to be supported is increasing with the advancement of wireless standards. Also, it is undesirable in the current times to spend a lot of time in arriving at a new architecture for being able to support enhanced error correction capabilities - in other words, the time-to-market for a new decoder design should preferably be less. For these reasons, it is preferable to develop architecture *templates* of *frameworks* that can accept the constraint length, code rate or generator polynomials as input parameters and yield efficient architectures for *any* value of these parameters. Several previous efforts have focussed on the development of such frameworks. (for instance, [72, 73, 13, 5]). The general architectural model used in these efforts is shown in figure 2.8.

These propositions aim to realize the required communication between the ACS units partly through the connections *between* the input and output local memories and the ACS units, and partly through the interconnection network connecting the memories. *Perfect shuffle* [50] or related networks are chosen for the interconnection between the memories.

As an example, the authors in [72, 73] propose to characterize the required communication pattern as a *matrix transformation*, which is realized through a sequence of three

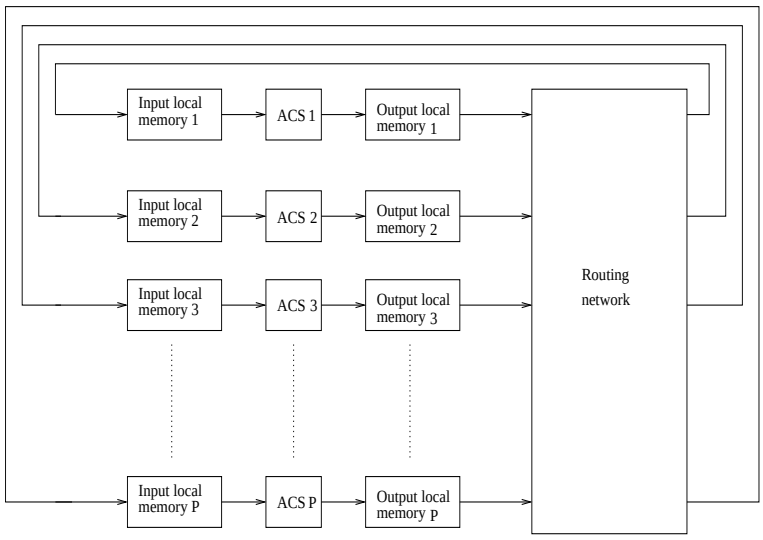
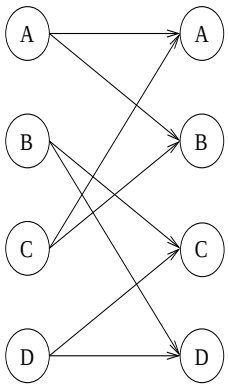


Figure 2.8: General architectural model used in [72, 73, 13, 5] which propose area-efficient architectural frameworks/families.

smaller matrix transformations - the first and third transformations only rearrange elements within the rows, while the second transformation only rearranges elements within the columns. The hardware is designed in order to support such transformations.

Interconnection topology: How efficiently path metrics and decision sequences can be transported between the ACS units is obviously affected by the *topology* used for connecting the ACS unit. Several previous efforts have investigated various different topologies for their usefulness towards Viterbi decoding. While hypercubes, cube-connected cycles and other networks have been considered in [18] for instance, de Bruijn and related networks have been considered in [57, 46]. A snapshot of the schemes proposed in [57, 46] is shown in figure 2.9. An important point to note here is that, [46] shows that the use of a de Bruijn interconnection network can support multiple constraint lengths without any change in network configuration.

These efforts show that *using de Bruijn and related networks can potentially lead to the derivation of efficient and scalable schemes for the communication between the ACS units.* Moreover, [57] shows that such networks can also be *laid out* in an *area-optimal* manner by following a regular approach based on the concept of *necklaces.* A necklace



PE	Time 1	Time 2
0	0000/0001	0010/0011
1	0100/0101	0110/0111
2	1000/1001	1010/1011
3	1100/1101	1110/1111

Figure 2.9: Four ACS unit architecture interconnected through a de Bruijn network implementing a 16-state decoder in the manner described in [46]. The table on the right shows the sequence of states scheduled on each of the ACS units. [57] basically describes the scheme used in each time unit of the overall scheme described in [46].

is formed by one complete cycle of nodes which are connected together by *shuffle* edges. For instances, in a 16-node de Bruijn network, the cycle of nodes $0 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1 \rightarrow 0$ forms a necklace. The proposed layout strategy places nodes within a necklace close together, with different necklaces placed in columns, and then shifts the positions of nodes *within* a necklace in such a manner that the *other* required interconnections in the graph (those not covered in the necklaces) can be provided efficiently and in a symmetrical manner.

Path metric and decision sequence memory access: There are basically two ways to partition the memory needed for storing the path metrics and decision sequences. In one approach, each ACS unit has a *local* memory which stores the path metrics and decision sequences required for its immediate ACS operation. In this case, it is necessary to achieve the required transport of data between the ACS units through the interconnection network connecting them. In the other approach, all the metrics are stored together in one or more *memory banks*, and all the ACS units access these banks for reading data as well as for writing results. The required transport of data is achieved by appropriate *addressing* of the memory banks. Quite obviously, as the number of ACS units increase, the requirements on the *bandwidth* of the centralized memory increases. As such, this

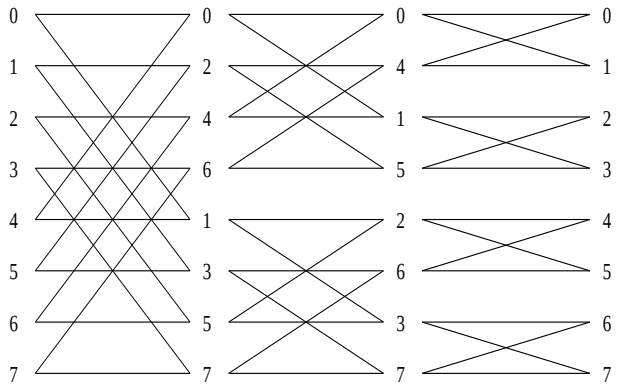


Figure 2.10: Locations of states and butterflies during one full cycle of in-place updating. Each row can be seen as a memory location, onto which different state metrics are written in different clock cycles. Note that the address of a given metric across successive stages can be obtained from its base address by a simple *right-shift* operation

approach is not easily scalable in order to support higher throughputs. Also, the size and power dissipation of the memory increases as its bandwidth is increased, further reducing the efficiency of the decoder. However, some previous efforts have yielded significant optimizations which ease the problem of the design and accessing of the central memory. Two such approaches are presently described.

While [8] proposes an *in-place* updating scheme which halves the path metric memory requirement, [47] proposes a technique called *swapped state grouping* for reducing the interconnection network to a *fixed* network of 2×2 switches, while still restricting the number of parallel accesses to a memory bank in each cycle to 1. The schemes are depicted in figures 2.10 and 2.11.

Pipelining the ACS operations: After a set of ACS operations are performed, it takes some number of clock cycles, in general, for the results to be routed to the ACS units where they are needed for the next set of operations. During these cycles, the ACS units need to be stalled. This leads to inefficient use of the available hardware. Hence, it is desirable to find methods of *pipelining* the ACS operations which are required to be performed, in order to increase the efficiency of the architecture. Two well-known methods for the same are presented in what follows.

A basic scheme for pipelining the execution of ACS operations *across successive*

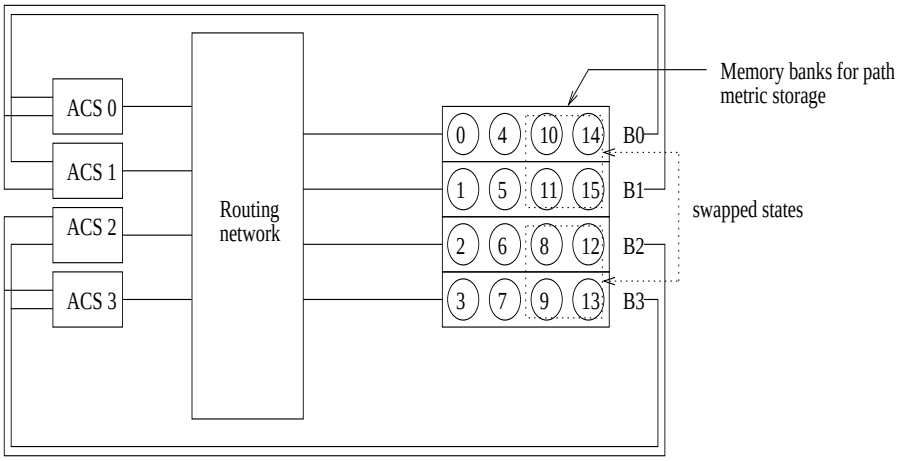


Figure 2.11: Metric storage scheme according to swapped state grouping for a 16-state Viterbi decoder. The architecture consists of 4 ACS units and as many memory banks. The locations of the metrics corresponding to the high half states (states 8 through 16) are *swapped* relative to the normal binary order of storing. Besides this, a special *assignment* of state computations onto the ACS units is also derived in order to restrict the access rate for each memory bank to 1 per cycle.

stages is proposed in [39]. An architecture with this kind of pipelining is called a *cascade* architecture. The cascade architecture proposed in [39], called the Canonic Cascade Viterbi Decoder, uses $log(n)$ ACS units, where n is the number of states in the decoder, and a *ring* topology to interconnect them in order to achieve a reduction in routing area (refer figure 2.12). Each ACS unit is responsible for the execution of *all* the ACS operations in a stage of the trellis. Also, each ACS unit starts performing the next ACS operation after forwarding its current outputs to the subsequent ACS unit in the ring. Thus, in this architecture, multiple *stages* of the trellis are executed on the hardware in a pipelined manner. For appropriate transfer of the state metrics, each ACS unit is connected to its neighbours through a shift register and a cross-point switch. This architecture has been improved in terms of efficiency in [35, 36].

Another method of pipelining the ACS operations *within* a stage is described in [22]. The technique is called *trellis pipeline-interleaving*. The ACS operations of the states in the trellis are shown to be *loosely coupled*, in the sense that, the trellis can be divided into subtrellises which can be executed *independently* most of the time and the results need to be merged only once in every $log(n)$ stages, where n is the number of states in the

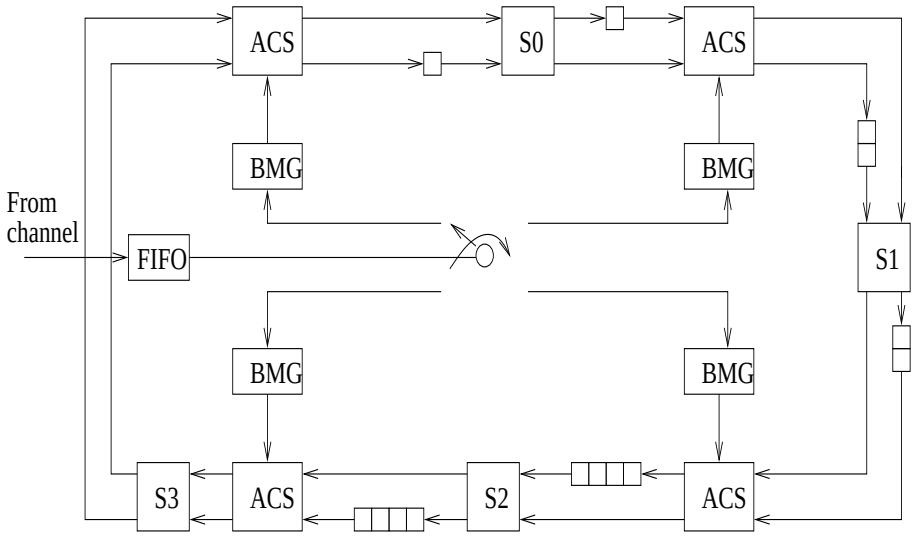


Figure 2.12: An example canonic cascade Viterbi decoder with a binary alphabet and $n = 16$ states. ACS = Add-Compare-Select. BMG = Branch metric generator. S = Switch. The squares represent storage elements.

decoder. This characteristic is brought out using the topological equivalence between the trellis (which is a shuffle-exchange network) and a binary *n-cube* network of equivalent size (refer figure 2.13). This technique thus identifies the states which need to be mapped to each ACS unit so that the ACS units can be deeply pipelined and kept active most of the time, except for a small number of idle cycles at the end of every $log(n)$ stages.

A common framework for obtaining both the kinds of pipelining described above is described in [7]. Pipelining methods known for FFT computation have also been applied to Viterbi decoding, as, for instance, in [45] and [65].

State-parallel approach

When the focus is on obtaining a *high speed* of decoding, having as many ACS units as the number of states becomes an attractive option. In this case, there are two basic problems which need to be tackled - (i) Optimal exchange of metrics between a large number of ACS units. (ii) Identification of additional sources of parallelism in order to use the provided hardware efficiently, as state-level parallelism is already fully exploited. A few existing approaches to these problems are described in what follows.

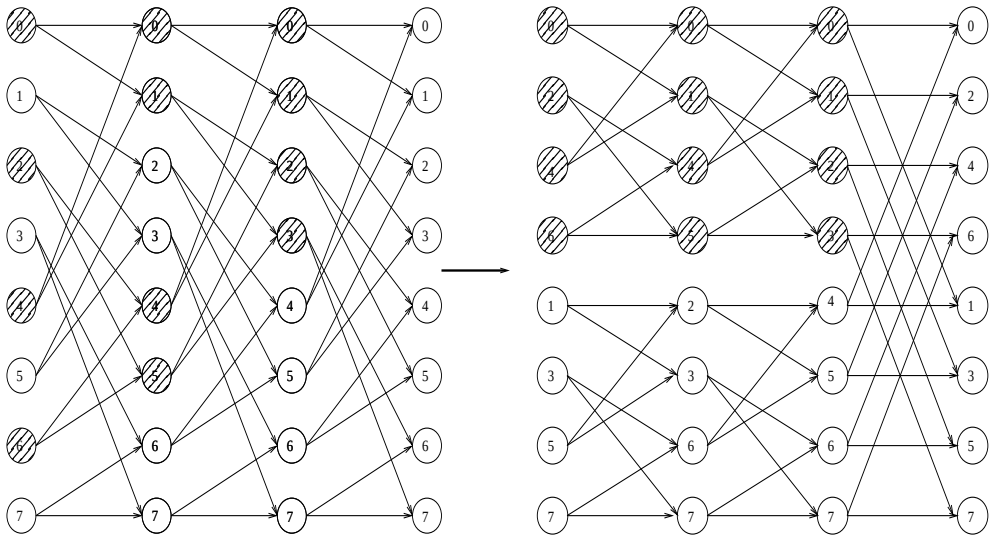


Figure 2.13: Splitting the recursion into loosely couple parts. The network on the right is equivalent to an n-cube network. The different shadings indicate the loosely coupled sets of computations.

A number of different interconnection networks have been studied for their usability in the transfer of path metrics between the ACS units in a fully parallel Viterbi decoder realization. Again, as in the case of state-sequential decoder, *shuffle-exchange* networks have been found to be favourable choices for connecting the ACS units in a fully parallel Viterbi decoder ([40, 41]). A two-column, *ring*-based topology based on the concept of *Hamiltonian* cycles which are found in de Bruijn graphs is proposed in [76]. This is analogous to the physical layout strategy proposed for the shuffle-exchange network in [57] (refer section 2.5.1.

In order to tap additional sources of parallelism, beyond that offered by the ACS computations of a single stage, it is necessary to look at parallelizing ACS operations across *multiple* stages. A similar idea was proposed earlier as the *cascade* architecture in section 2.5.1 (refer figure 2.12. [10] proposes a different way of extracting this parallelism - by *combining successive stages of the trellis into a single stage.* (refer figure 2.14). This modification increases the number of inputs and outputs (in other words, the radix) for the computations at each state. Although this increases the hardware complexity, the achievable throughput is also increased. [10] shows that the relative increase in

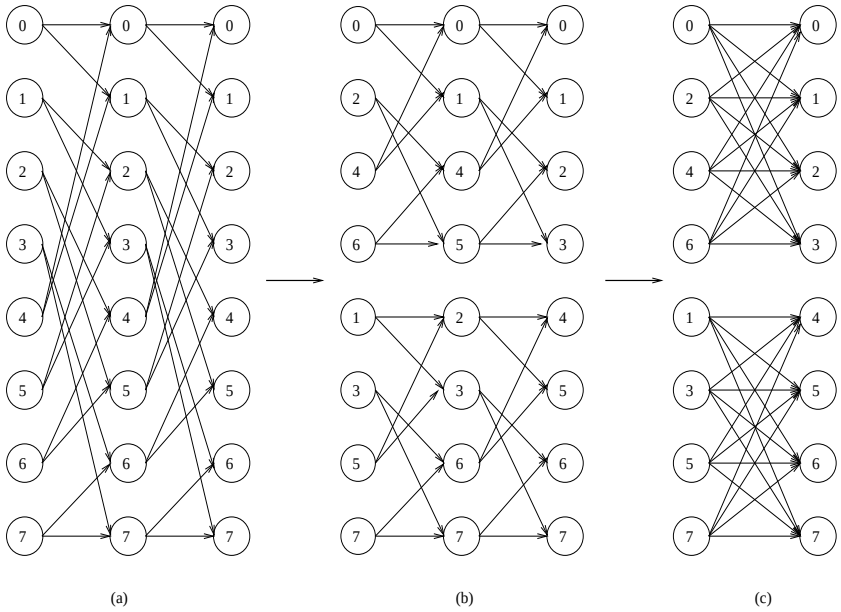


Figure 2.14: (a) 8-state radix-2 trellis. (b) 4-state subtrellis decomposition. (c) 8-state radix-4 trellis. A radix-4 trellis represents the case when computations across *2* successive stages are combined into a single stage.

throughput is more than the relative increase in hardware complexity and silicon area, which makes the realization of higher-radix trellises an attractive option.

Many other methods of extracting parallelism in Viterbi decoders have been proposed in earlier literature. Some of them have been covered separately in section 2.5.4.

Bit-serial arithmetic, as an approach to reduce power dissipation and area in fully parallel Viterbi decoders has been proposed by Chang *et al.* in [17].

2.5.2 Optimal representation of path metrics

As the Viterbi algorithm proceeds through the stages, the path metrics go on increasing in value. If the maximum size of a unit information packet used in the communication standard is known, the bit-width required for path metric representation can be decided accordingly. However, in practical cases, this maximum size is relatively large. Also, there are several ACS units in most Viterbi decoder implementations. Hence, it is of great interest to see if the required bit-width for the path metrics can be reduced by the use of advanced schemes for path metric representation. Intuitively, we can consider

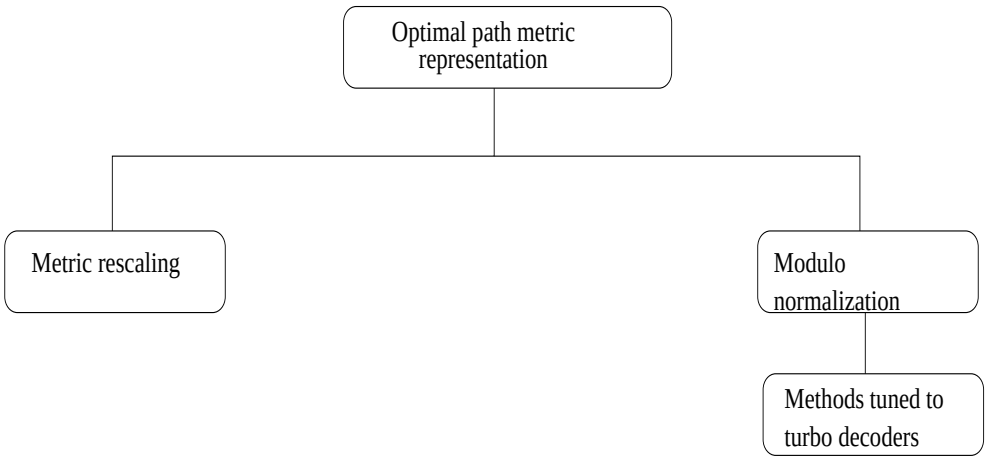


Figure 2.15: Broad subdivision of previous research related to path metric representation in Viterbi decoders.

subtracting a common quantity from *all* the path metrics at regular intervals in order to limit the bit-width used for the path metrics, while ensuring that the comparison results are not affected at any point. This is the basis of the *metric rescaling* approach to limiting the path metric bit-width. The bit-width reduction is proportional to the frequency of rescaling the metrics. Such methods can be found in literature dated prior to 1989 (for instance [19, 39]). The other approach, which is now the most exclusively used method, is called *controlled overflow*, proposed by Hekstra in 1989 [43]. This method was known by subject experts prior to this publication, but it was documented in publicly available literature for the first time by Hekstra. The great advantage of this method is that *no* additional circuitry (like the subtractor needed in the metric rescaling scheme) is needed to obtain the bit-width reduction. The key idea is to accommodate the overflow which will occur due to the finite bit-width in a controlled manner by incorporating *two's complement arithmetic*. A slight increase in bit-width is necessary in order to maintain the correctness of the decisions. The method is based on two properties of the Viterbi algorithm:

1. The output of the algorithm depends only on the differences of the path metrics.

2. The difference between metrics is *bounded*.

Assuming a 2's complement number representation, the movement of the path metrics is as follows. Path metrics start from 0 and go on increasing till they reach the maximum value that can be represented by the number system, at which point they *overflow* going to the most negative number which can be represented in the number system. From here, they increase in value again, until they cross 0 and the entire process repeats over and over again. The first property above allows us to compare the 2's complement values of the path metrics since the output of the algorithm does not require the *actual* values of the path metrics to be retained, but only their relative differences.

However, in such a system, additional care is required in order to ensure that the comparison results are correct at all times. To identify the cases where ambiguity could occur, let us consider the process of comparison of two arbitrary path metrics (a and b) represented according to the 2's complement representation. The most direct comparison is an *unsigned* comparison of a and b. However, since the *number of times* a path metric has circulated over the number range is not recorded, the result of an unsigned comparison will not be necessarily correct. For instance, suppose $a < b$ by direct comparison. But if a has overflowed more number of times than b, then actually, $a > b$. This ambiguity can be removed if one can guarantee that the *largest* path metric will not *reenter* a given half of the number system until the *smallest* path metric exits that half. If this condition is guaranteed, the comparison becomes easy as the unsigned comparison will always yield the correct results. A very useful analogy for the behaviour of the path metrics is given in [74], which is reproduced briefly here. The accumulating path metrics can be viewed as runners in a race, involving *laps* around the range of the number system. The condition stated above, that the *largest* path metric does not *reenter* a given half of the number system until the *smallest* path metric exits that half indicates that it is a very competitive race. In other words, no two runners in the race are separated by more than half a lap. In this condition, the relative placement of the "runners" can be derived unambiguously from their current absolute locations on the track, even without knowledge about the current lap number. This is the basic idea of the scheme proposed in [43]. The range of the number of system is decided by the

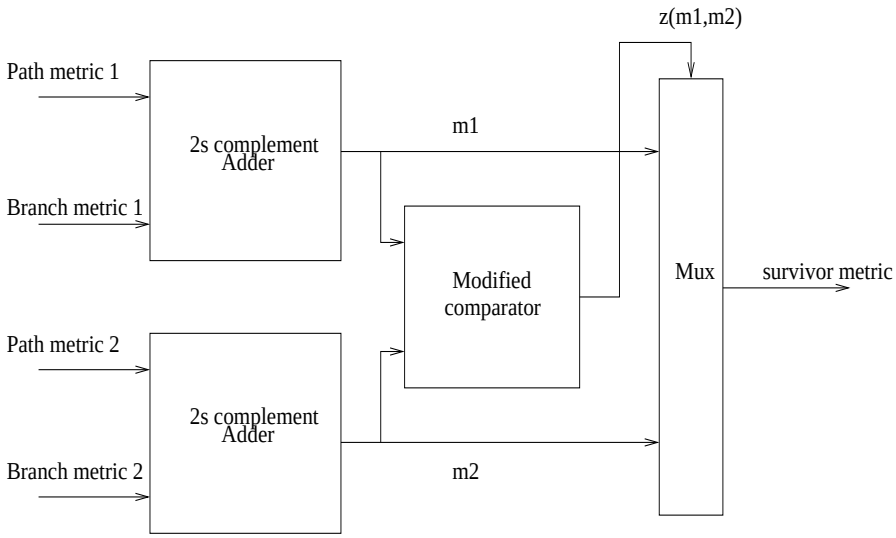


Figure 2.16: The modulo normalization architecture of [74] using the *modified comparator* rule, which can be stated as follows: Assume that y(m1,m2) is the result of an *unsigned* comparison between the quantities (m1)%(P) and (m2)%(P), where P=2^x and x is the number of bits used for path metric representation. Then, z(m1,m2) equals y(m1,m2) or its complement depending on whether m1 and m2 are of the same sign or of opposite signs (in other words, depending on whether the MSBs of m1 and m2 are same or different).

bit-width used for the representation of the path metrics. The condition described above requires that the range of each *half* of the number system be at least p, where p is the maximum difference between the path metrics (which is bounded). Hence, compared to a direct representation of the path metrics, a cost of *one* extra bit is involved in using the controlled overflow scheme.

[74] also reviews various other normalization schemes, concluding with the inference that the controlled overflow method is the most favourable. It also proposes a *modified comparison rule* which reduces the complexity of the ACS circuitry when compared to a circuit which directly implements the scheme described in [43]. This architecture is used in most contemporary implementations. The architecture is depicted in figure 2.16.

For turbo decoders, the behaviour of the path metrics change due to the feedback nature of communication between the two decoders constituting the turbo decoder. Normalization schemes similar to [74], but geared to turbo decoders have also been proposed in literature ([84, 85, 38, 55] for instance).

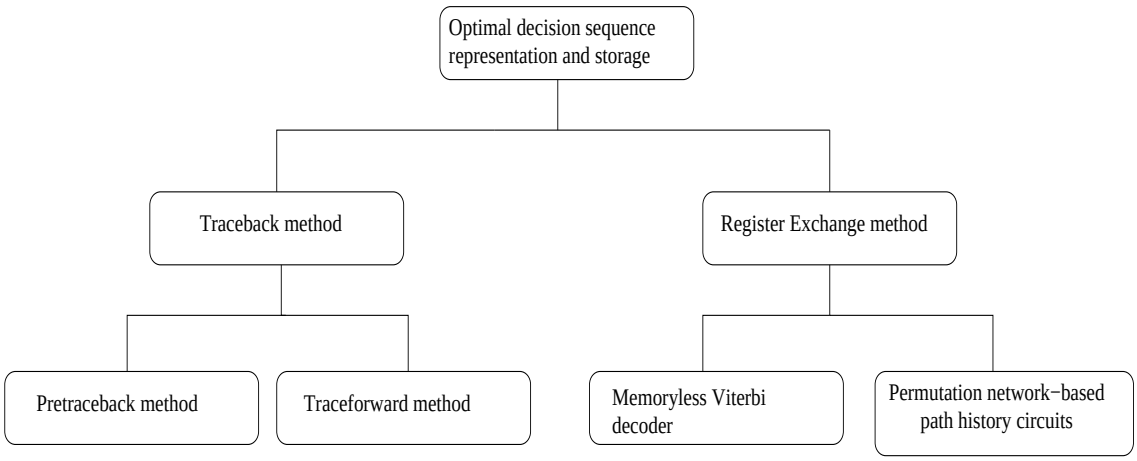


Figure 2.17: Broad subdivision of previous research related to optimal survivor path representation and management.

2.5.3 Optimal survivor path representation and management

At regular intervals between the ACS operations, it is necessary to obtain one of the *survivor paths* from the stored decision bits, off which one can then read the decoded output bits. The most straightforward method to do this is by maintaining for each state, a register which contains, in terms of decision bits, the complete survivor path passing through that state back a certain number of stages which is at least equal to the truncation length. At each state, for each stage, the path with the lower accumulated path metric is identified through the ACS operation. Then, the entire path information is transferred from the register corresponding to the *previous* state on this path to the register corresponding to the current state, alongwith the required updation with the newly obtained decision bit. This method is called the *register transfer* method and is depicted in figure 2.18.

This method requires a huge word (containing the complete path - which can be 50 bits long for a constraint length 9 decoder according to the 5 times constraint length heuristic) to be transferred between the processing nodes at each stage. This considerably increases the interconnection complexity and also increases the power consumption. However, obtaining the decoded bits does not need a separate process as the decoded bits can be read off the appropriate locations of any randomly selected register. The

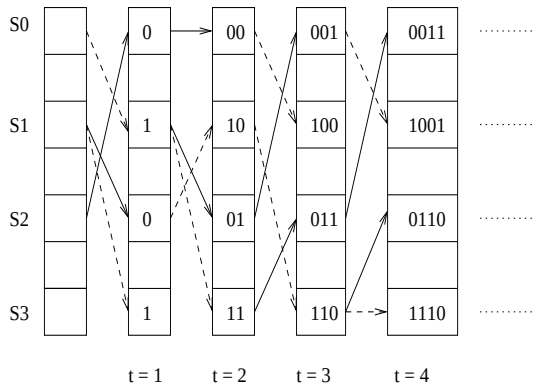


Figure 2.18: Basic register transfer method shown for a 4-state Viterbi decoder. The figure shows the updation of the registers associated with each of the four states with the passage of time. The bit appended to a path depends on whether the most recent transition occurs due to an input bit 0 (indicated by a normal arrow) or an input bit 1 (indicated by a broken arrow).

register selected for reading can be randomly selected due to the merging property of the survivor paths explained earlier in section 2.3.

[68] proposed the *other* basic method in 1981, called the *traceback* method. In this, at each stage, for each state, *only* the decision bit obtained from the ACS operation of that stage is stored. As the bits are stored across a certain number of stages which again, is at least equal to the truncation length, the total memory requirement remains the *same* as for the register transfer method. However, the interconnection complexity is greatly reduced as the complete path information is not transferred between the processing nodes. The penalty for the reduction in interconnection complexity is paid for through increased latency for obtaining the decoded bit, as now, a *separate* process is needed to explicitly *trace back* the survivor path. The basic step to be carried out at each step of the traceback process is obtaining for a selected survivor path, the state at the *previous* stage, from the state on the survivor path at the current stage and the current decision bit. Suppose that S_t is the state on the survivor path at time t and d_t is the current decision made at time t. Then, the state S_{t-1} on the survivor path at time $(t-1)$ can be obtained as: $S_{t-1} = [(S_t << 1), d_t]$, where $[a, b]$ denotes the concatenation of two bit fields a and b. The direction of shifting of S_t in the traceback process is opposite to that in the encoder. The particular type of shift (left or right) is only a matter of convention.

Here, a left shift is shown in order to correspond to the right shift of the current state performed in the encoder of figure 2.1.

This process continues until the state on the survivor path at time $(t - L)$ is found, where L is the truncation length. From that point, the information bits associated with the transitions traced back can be directly provided as the *decoded* bits.

A variety of alternatives to these two basic methods have been proposed in the literature. A few of the important ones are described below in the order in which they were proposed. The aim is to provide an idea of the state-of-the-art in this well-researched sub-topic within the area of Viterbi/turbo decoder design.

Intuitively, the throughput of the traceback method can be increased by reading and processing *more than one* decision bit at a time. This basic idea is proposed by Shung and Cypher, in 1993 [20]. The authors show that having multiple read pointers allows for a reduction in the speed of read access, reduces the memory requirements and allows for a uniform speed realization, which requires only one clock. General equations for the multiple read-pointer technique are derived and two implementations based on standard RAM and custom shift registers are presented.

Black and Meng, in 1993 [11], proposed two new survivor management techniques, which are hybrid versions of the two classical techniques, register exchange and traceback. The first, called *pretraceback*, involves adding some levels of *lookahead* to the traceback recursion. Briefly, single-stage decisions over a certain number of stages are stored together as composite decisions, which allows multiple stages to be traced back with a single traceback operation. This speeds up the traceback process which increases the achievable throughput. The pretraceback is implemented through a register transfer network with length equal to the number of stages whose decisions are collated as composite decisions (the number of levels of lookahead). The second technique proposed, called *traceforward* eliminates the need for a traceback recursion to find the starting state for the decode operation by obtaining and storing the decode starting state for each possible path during the *forward* ACS update. This again helps in reducing the number of cycles needed to perform the traceback operation, albeit at the cost of extra

logic and storage for identifying the *tail* state for each surviving path. The pretraceback method is shown to be more advantageous for radix-2 trellis of more than 16 states, and the traceforward method advantageous for trellises of fewer states but a higher radix.

Lin, in 2000 [53], proposed a new method for traceback, in which each node only stores the decision bit corresponding to its cycle similar to traceback, but these bits are used to *switch* a demultiplexer-based *permutation* network, which mirrors the underlying trellis. This leads to an architecture which consumes lesser area than the register exchange method, while still allowing the traceback to finish in one clock cycle, as the permutation network is combinational in nature. The paper also shows how a performance-routing area tradeoff can be achieved in this method by *folding* the permutation network over some number of levels.

El-Dib and Elmasry have recently proposed two innovations to the traceback process. In 2004 [24], they proposed the use of the "pointer" concept for easing the use of the register exchange method for large constraint length Viterbi decoders. The main idea is that a pointer is associated with the survivor path register for each state, and instead of transferring entire register contents as in the conventional register transfer method, only the pointer to the destination register is altered to point to the source register, which in effect, relabels the source register as the destination register. In 2005 [25], the same authors proposed a *memoryless* version of the traceback subsystem. This is an extension of the pointer-based technique in [24] with the additional constraint that the encoder be *reset* to a fixed state after every L iterations, where L is the truncation length.

2.5.4 Methods of extracting additional parallelism

To obtain very high throughput rate Viterbi decoders, additional sources of parallelism, other than the inherent parallelism available within the execution of the states of a single stage need to be exploited. A few schemes addressing this issue are described below. Again, a chronological order is followed so that the derivation of the more recent and more efficient schemes can be brought out.

Lin and Messerschmitt, in 1989 [51], have proposed several schemes for obtaining

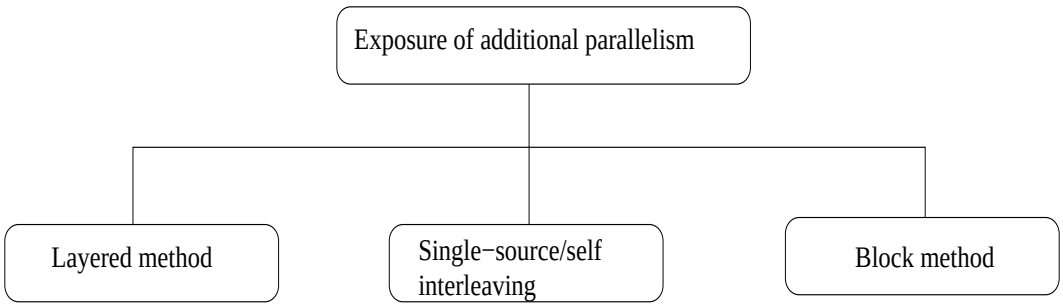


Figure 2.19: Broad subdivision of previous research related to exposing additional parallelism for high speed operation in Viterbi decoders.

unlimited parallelism (or equivalently, concurrency) in the Viterbi decoding process. The following methods are proposed:

1. *Layered method:* A method of decoding multiple trellis stages concurrently. The trellis is *collapsed* into higher order trellises ("super-trellises"), and higher level survivor path history at the level of the super-trellises is maintained and used in order to obtain higher decoding speed. Multiple layers of higher-order trellises may be used. (refer figure 2.20).

2. *Pipeline-interleaving:* Obtaining additional concurrency by interleaving the transmission of *multiple* information sources. (refer figure 2.21).

3. *Single-source/Self-interleaving:* Obtaining additional concurrency by interleaving the transmission of information bits (out-of-order transmission) from the same source. The code performance is, however, impacted by this type of interleaving, which places limits on the amount of interleaving possible.

4. *Block method:* Obtaining additional concurrency by dividing the information sequence into blocks where the encoder memory is *reset* at the start of each block. In this case, the starting encoder state for each block is known *a-priori*, due to which these blocks can be decoded concurrently.

When the information source is *controllable* (ie. extra bits can be inserted into the information bit sequence), the block method is shown to be the most attractive

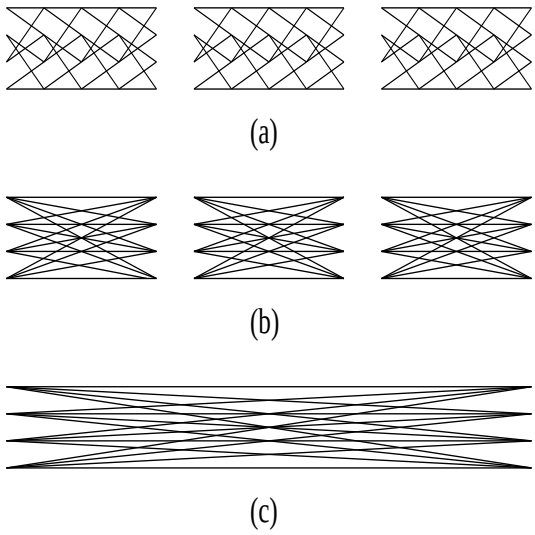


Figure 2.20: An example trellis at (a) layer 1 (b) layer 2 (c) layer 3

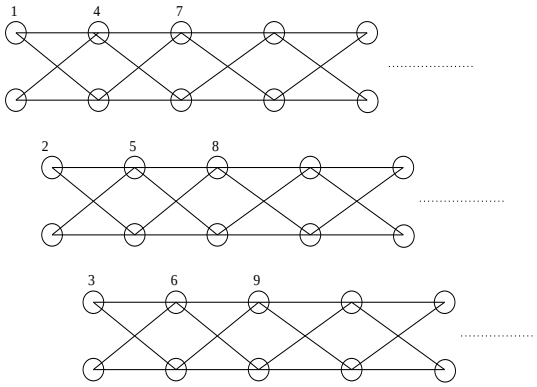


Figure 2.21: Pipeline interleaving of *three* information sources. The number on a node shows the information bit which is processed in that stage.

alternative, while the layered method is shown to be the best method when the source is not controllable. These techniques have been generalized for arbitrary finite state machines (FSMs) by the same authors in [52].

Fettweis and Meyr have explored a number of alternatives for obtaining additional parallelism which have been widely cited in subsequent literature. A survey of their work until 1991 is available in [34]. Three levels at which parallelism can be introduced are identified - bit level, word level and algorithm level. The innovations are listed below:

1. For bit-level parallelism, the use of *carry-save* adders, rather than ripple adders is proposed in order to obtain a reduced critical path length. This work is also described in [32]. This modification also allows pipelining to be introduced into the adders which leads to an architecture whose achievable throughput is independent of word length [30].

2. For word-level (stage-level) parallelism, the algorithm is algebraically reformulated to expose a very useful fact - that the addition and maximum selection operation which constitute the ACS operation form an algebraic structure called a *semi-ring*. The authors employ the fact that equations formed with semi-ring operations are linear, and apply *lookahead* computation techniques which are applicable to linear equations, in the context of Viterbi decoders. This increases the achievable throughput. More simply put, this property makes it possible to combine the execution of multiple stages of the trellis into a single operation. (This is similar to the radix-4 operation described in [10]). This work is also described in [32, 29] and by other authors in [78, 83]. The method can also be viewed as a special case of the *layered* method with *2* layers, as noted in [51].

3. For algorithm-level parallelism, the following observation is used. If a decoder starts decoding in the mid-stream of the data ie. somewhere in the middle of the trellis, with arbitrary starting path metrics, a *limited* number of stages of initial synchronization occurs after which the decoder behaves exactly as if it had started decoding at the start of the trellis. In fact, this synchronization can be shown

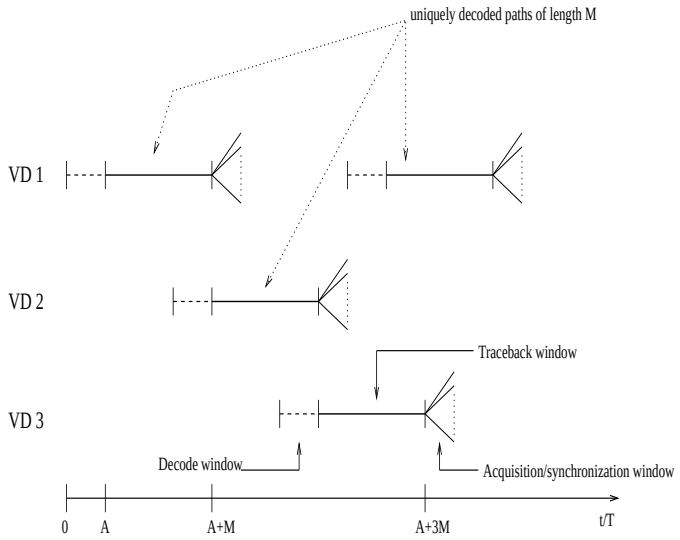


Figure 2.22: Scheme of the *overlap-abut* method of block decoding

to complete in the same number of stages as the truncation length [33]. Hence, it is possible to break up the incoming data into multiple *blocks* and perform the ACS operation within these blocks in parallel. For each block of length M symbols, apart from the M ACS stages which are anyway needed, an overhead L ACS stages *prior to* and L ACS stages *after* the block is necessary to obtain the correct results. This idea leads to a family of new parallel architectures as described in [33], capable of achieving throughputs which are higher than conventional decoders by orders of magnitude. The scheme used in one of the architectures in this family is depicted in figure 2.22.

4. Another important innovation is the combining of the word-level and algorithm-level parallelism into a single architecture. This has been documented as the *minimized* method [28, 31].

The basic idea of algorithm-level parallelism is described earlier in [79] (1981). The authors additionally proposes a ROM-based table lookup implementation which is infeasible for any practical decoder size, and hence not followed up in subsequent literature. The same idea is also independently proposed in [9], in addition to an important observation: The encoder state reinitialization methods of [51] cannot be applied to the

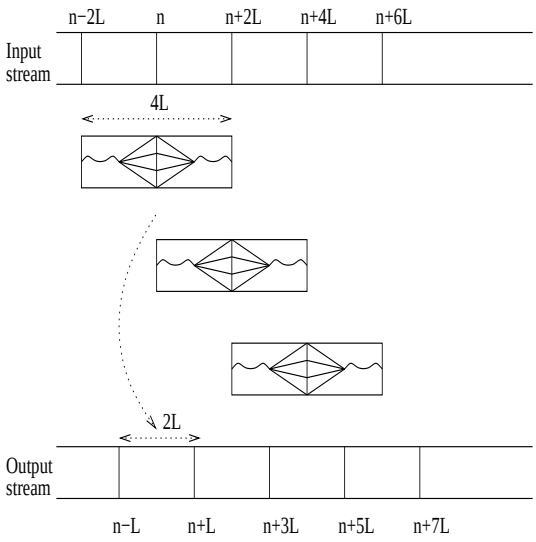


Figure 2.23: Continuous stream processing using the (Sliding Block Viterbi Decoder (SBVD) method proposed in [12].

problem of digital sequence detection in the presence of *intersymbol interference (ISI)*.

Black and Meng, in 1997 [12], have generalized the sliding block method of [79]. The scheme proposed by them is shown in figure 2.23. Note that, if all the received symbols are available, the path metric updation process can be performed either in the forward direction (left to right) or in the backward direction (right to left). In [12], a hybrid algorithm is described which allows for a *combination* of forward and backward processing of the input data blocks (with appropriate merging of paths somewhere in the middle) and the forward-only and backward-only methods are shown to be special cases of this algorithm. To increase the throughput, a block of data is decoded through a *set* of forward and backward passes, which allows the identification of the path with the minimum accumulated metric. As this path is known to be the most likely path (not randomly selected), *all* the information bits associated with the block can be decoded in a single step. It is shown that the survivor path length in the proposed method can be reduced to *2.5* times the constraint length since it is a *best-state* survivor path decoding algorithm. Finally, in order to obtain even higher throughput, a *systolic* implementation of the hybrid method is described which combines the algorithm-level concurrency of the Viterbi algorithm with pipelining of the various sub-units of the decoder.

2.6 Flexibility requirements revisited

Here, we provide a brief description of the requirements imposed on channel decoders by modern wireless standards in the context of the Viterbi/turbo decoding algorithms. This is necessary for placing the contributions of the thesis in context.

The higher the constraint length and code rate, the more robust the error correction, which is intuitive because the effect of a single information bit is spread over a larger number of output bits. Hence, for modern wireless standards, the constraint length as well as the code rate is allowed to *vary* during the course of a conversation. Accommodating these two kinds of flexibilities leads to requirements on distinct sections of the decoder hardware. The code rate flexibility needs to be built *into* the ACS units, by appropriately modifying the incremental/branch metric calculator unit. This inference follows directly from knowledge of the operation of the Viterbi algorithm. The constraint length flexibility needs to be built into the *interconnection network* connecting the ACS units. This can be derived from the fact that for a given constraint length K, the communication pattern required for the transfer of data (path metrics) generated at each stage can be expressed as a *de Bruijn* graph of 2^{K-1} nodes (or states). Different values of K would thus require the updated path metrics to be transferred along de Bruijn graphs of different sizes.

Chapter 3

De Bruijn network-based architectures

In chapter 1, it was shown that the code rate flexibility places demands on the *incremental metric generator* of the ACS unit, while constraint length flexibility places demands on the *interconnection network* connecting the ACS units. This thesis focusses on providing constraint length flexibility in an efficient manner. In this chapter, firstly, a brief survey of recent results related to *modern/flexible* channel decoder implementation is provided, and the approach taken in this thesis is contrasted with respect to these results. Subsequently, two approaches to building *flexible* constraint length Viterbi decoders on a *de Bruijn* interconnection network are presented. In the first approach, the problem is viewed as a *graph embedding* problem, and mathematical manipulations are presented in order to achieve a "good" embedding, with respect to certain parameters. In the second approach, a more hardware-centric approach is taken and another scheme is presented. Although the first approach leads to an answer for the question of *how well* smaller de Bruijn graphs can be embedded on a larger de Bruijn graph (which has not been attempted earlier to the best of the author's knowledge), the synthesis results in chapter 5 show that the second scheme leads to a more *area-efficient* realization of flexible constraint length Viterbi decoders on a de Bruijn interconnection network.

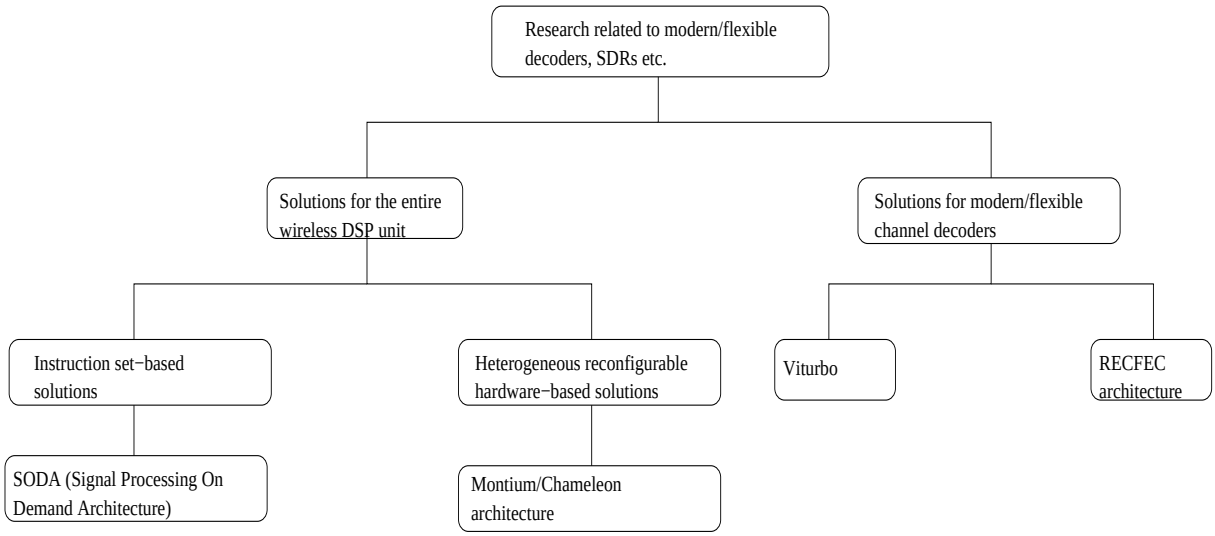


Figure 3.1: Broad subdivision of previous recent research related to the design of modern/flexible channel decoders and their application to SDRs.

3.1 Previous results related to modern/flexible channel decoder implementation

As seen earlier in chapter 2, plenty of research has already been done in tackling the basic issues related to the implementation of Viterbi/turbo decoders. In recent times, the focus has slightly shifted from such basic issues, to the design of more efficient *flexible* channel decoders as also towards architectures which are capable of supporting *all* the functionalities in the wireless DSP unit *together* with the channel decoder, without inordinately compromising on performance and power/energy dissipation. This section provides a brief survey of existing work related to such architectures.

3.1.1 Architectures for the entire wireless DSP unit

Recently proposed architectures for this purpose can be divided into two main types -

1. *Processor-like* solutions, where a combination of pipelining and other techniques like Single Instruction Multiple Data (SIMD)/Very Long Instruction Word (VLIW) are used to exploit the *parallelism* inherent in many of the algorithms required in the wireless DSP unit. Examples of such architectures are [80, 54, 27].

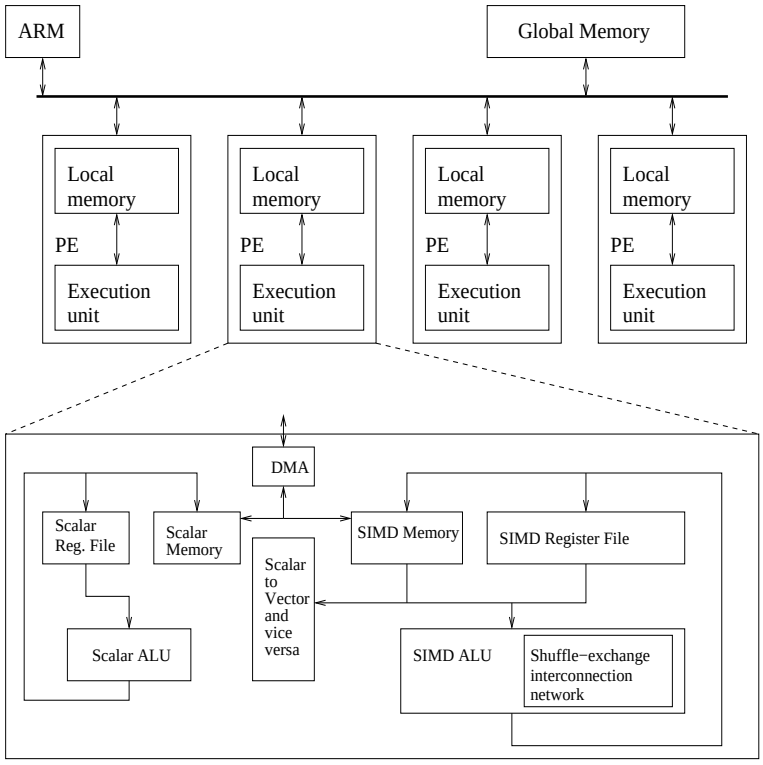


Figure 3.2: High-level block diagram of the SODA multicore DSP architecture.

2. *Heterogeneous reconfigurable hardware* solutions, where individual nodes are customized for handling the different types of computing loads presented by the algorithms required in the wireless DSP unit. The nodes are often interconnected through a *Network-on-Chip (NoC)*. Examples of solutions in this class are [64, 69, 70, 2, 1, 4, 3].

As an example of the first type, we describe in some detail, the architecture proposed in [54]. This architecture is called *Signal-processing On-Demand Architecture (SODA)*. A block schematic is presented in figure 3.2.

The authors draw a few important conclusions regarding the nature of tasks in the wireless DSP unit:

1. The inter-kernel traffic, where kernels refer to the basic computational blocks in the wireless DSP unit, requires relatively *low* throughput. This suggests the use of less complex interconnection networks to connect the nodes, in order to reduce

the system complexity. However, the *type* of traffic between different kernels can vary widely. Hence, support for variable *buffer* sizes is necessary.

2. There are certain tasks (like FFT and channel decoding) which exhibit a huge amount of parallelism, which makes dedicated computing structures for such tasks desirable. (It can be seen from the description of the Viterbi algorithm in chapter 1 that there is no data dependency between the computation of the ACS operation for the states within a stage - in other words, these computations can all be performed in parallel. Additional sources of parallelism are also available in the Viterbi algorithm as will be explained in section 2.5.4.

3. 8- or 16-bit wide lines are sufficient for communication between various kernels. This relatively low bit-width eases the requirement of data movement within the SIMD units.

As can be seen in figure 3.2, this architecture consists of multiple processing elements (PEs), a scalar control processor (ARM) and a global scratchpad memory, all connected through a shared *bus*. Each PE consists of a scalar unit and a wide SIMD unit. The SIMD unit runs most of the compute-intensive algorithms, like searcher, FFT and the channel decoder. The scalar unit is used to support many of the DSP algorithms that are scalar in nature and cannot be parallelized. The processing model used can be defined as *VLIW (between scalar and SIMD units) + SIMD*, with *static* scheduling of kernels on the PEs. One notable point is that a *shuffle-exchange* network is used to connect the individual processing components *inside* the SIMD unit, as it is found to fit well into the communication requirements as laid down by the various tasks which have to be carried out on the SIMD unit.

As an example of a heterogeneous/coarse-grained solution for the wireless DSP unit, we describe the *Montium* architecture proposed in [70]. The overall fabric, called *Chameleon*, is a heterogeneous architecture, where individual nodes could be DSPs, embedded FPGAs, ASICs, GPPs or the Montium processor, which is a *coarsely* reconfigurable custom processor meant to support the computational demands of typical DSP algorithms. The

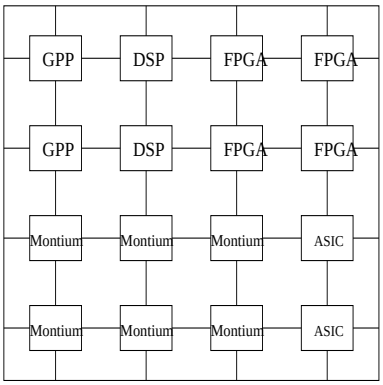


Figure 3.3: Chameleon SoC template

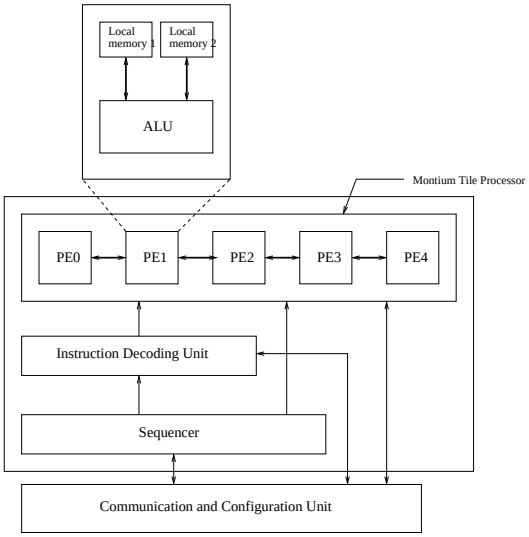


Figure 3.4: Montium processing tile

nodes are connected together by a *Network-on-Chip (NoC)*. The proposition is to choose the *granularity* of individual nodes according to the set of operations required in a given application (for instance, SDR). A high level block diagram of the Chameleon SoC template is shown in figure 3.3. The authors also propose a new node architecture called *Montium*, which can provide efficient support for algorithms having different amounts of inherent parallelism. A high level diagram of the Montium tile is shown in figure 3.4.

The Configuration and Communication Unit is responsible for configuring the set of five ALUs (with memory) comprising the *Tile Processor (TP)*, as well as providing support for both streaming and block processing modes in the TP. Configuration information includes the instructions to be executed on the TP and the interconnection

pattern to be implemented between the individual processing elements in the TP. The configuration information is provided through a *file* loaded onto the internal memory of the Montium tile through the NoC connecting the heterogeneous processing nodes. After the datapath and instructions have been set up as necessary within the TP, the *Instruction Decoder* and the *Sequencer* blocks orchestrate the execution of the required operations on the TP.

Research efforts have also been directed towards optimal design of the NoC connecting such heterogeneous processing nodes (for instance, the *Scalable Communications Core (SCC)* [44] and the Globally Asynchronous Locally Synchronous (GALS)-based *FAUST* NoC [49]). The GALS design methodology allows for different *clock domains* to be used for different regions within a complex System-on-Chip (SoC) which greatly eases the design of the clock distribution network.

3.1.2 Architectures for modern/flexible channel decoders

Chapter 1 described how modern wireless standards (3G and above) demand a high amount of *flexibility* in the operation of the channel decoders. From here on, we review work related specifically to the design of channel decoders which are flexible in the set of supported parameters as well as in the set of supported decoder types.

Studies carried out in [48] and [23] have recently shown that it is most beneficial to combine Viterbi and turbo decoders or turbo and LDPC decoders. These studies also show that the maximum savings are achieved by sharing the *memory* resources between the decoders. One of the earliest architectures proposed for *reconfigurable Viterbi/turbo decoders* is the *Viturbo* architecture [15]. It is capable of supporting Viterbi decoding with a code rate of either $\frac{1}{2}$ or $\frac{1}{3}$ and constraint length anywhere between 3 and 9. Viturbo can also support turbo decoding with a code rate of $\frac{1}{2}$ and constraint length of 4. A high level view of the Viturbo architecture is shown in figure 3.5.

The authors adopt a fully parallel approach for performing the ACS operations, with an ACS unit for each state of the decoder across all constraint lengths. The *Branch Metric Unit (BMU)* calculates *all* possible branch metrics for a decoder with a specific

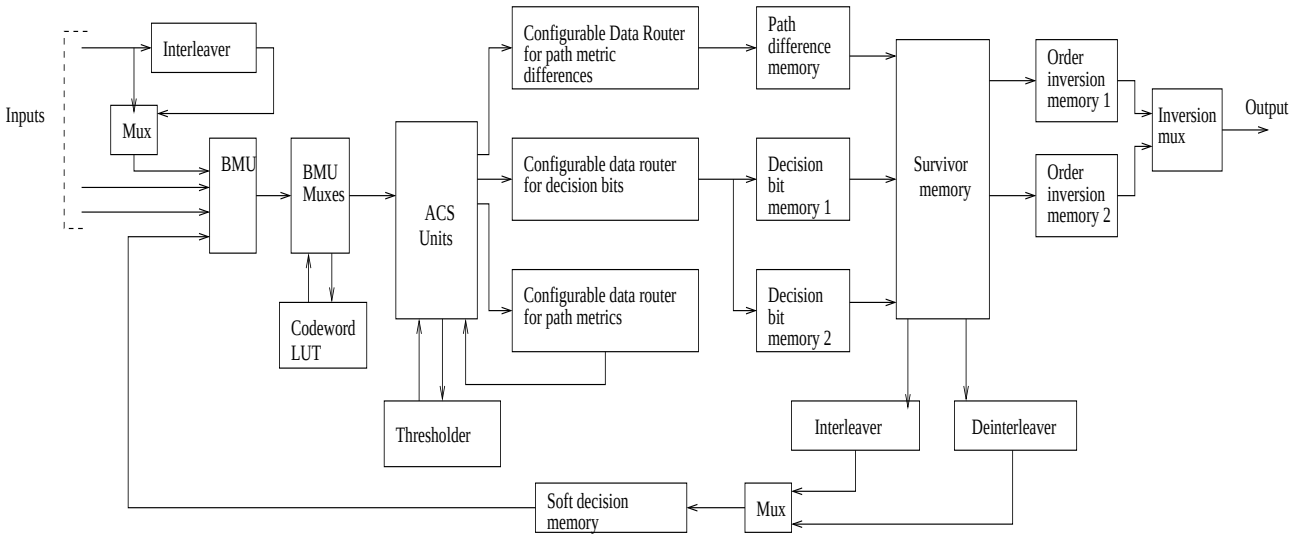


Figure 3.5: Viturbo - High level block diagram. BMU: Branch Metric Unit. LUT: Look-Up Table. ACS: Add-Compare-Select.

constraint length and rate, based on the received symbols. Specific sets of branch metrics are needed for each ACS unit to perform the ACS operation. These metrics are selected and routed to the ACS units by the *BMU muxes* together with the *Codeword LUT*.

As shown in figure 3.5, the problem of routing path metrics is solved by using a multiplexer-bank based *configurable data router*, with *custom* logic to support multiple constraint lengths. Finally, for power reduction, custom switch-off logic has been used to power down those parts of the system which are not required for a given type and size of decoder.

More recently, an instruction set-based architecture called RECFEC (REConfigurable processor for Forward Error Correction) has been proposed in [63]. A high level block diagram is shown in figure 3.6.

During normal operation, *Configuration Words* which accommodate the configuration information of PEs are broadcast from the *Configuration Buffer (CB)* to the *Processing Element (PE) Pool*. The *Data Buffer (DB)* is embedded data memory that interfaces with external memory and pumps data to the PE Pool. A high throughput data network connecting the PE pool, DB and CB facilitates the supply of data to and collection of the results from the PE Pool (shown by arrows in figure 3.6). No particular interconnection

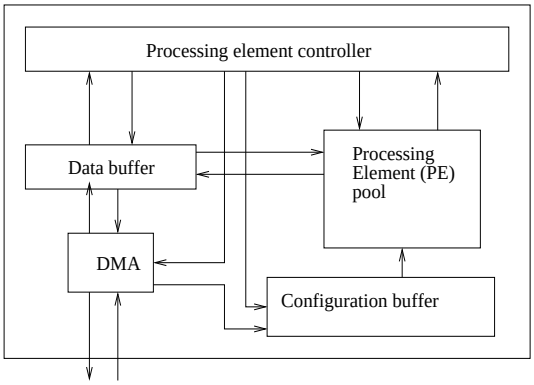


Figure 3.6: RECFEC architecture

topology has been adhered to. Reconfiguration of the network connecting the DB to the PE Pool facilitates the support of different data movement patterns. All data transfers between the DB or the CB and the *external* memory are handled by the *DMA Controller*. The PE Controller is a general purpose 32-bit RISC processor which controls the sequence of operations. Each *Processing Element (PE)* in the PE pool contains circuits used for implementing the various types of decoders as well as an ALU for performing basic logic and arithmetic functions. The sequence of operations and the configuration words are specified through *instructions*. Due to this property, this architecture is able to support a wide range of decoding types, and is also truly *software upgradable*.

Using FPGAs to implement the required flexibility has been investigated by Tessier *et al.* in [77]. The *adaptive* Viterbi algorithm [75, 16] is used for decoding in order to reduce the power consumption, which otherwise, is higher for FPGA-based solutions when compared to ASIC-like solutions. The adaptive Viterbi algorithm is an extension of the conventional Viterbi algorithm where some of the survivor paths are *pruned* at each stage based on the magnitude of difference between their path metrics and the minimum path metric at that stage. The authors show that significant power gains are to be made by *dynamically* reconfiguring the decoder constraint length according to the channel conditions. A detailed study of the obtained performance and the power consumption of the implementation of a flexible decoder on an FPGA is provided. The throughput, however, is limited to the range of hundreds of Kbps, while the power

consumption is of the order of hundreds of mW. Another approach has been described by Campos and Cumplido in [14]. The main innovation performed by the authors is the design of a more compact ACS unit, where the Compare operation is performed first, followed by the Select and finally the Addition. The throughput quoted is higher than that in [77]. However, a comparison is difficult since different FPGA boards have been used. Also, the power consumption is not mentioned. Note that neither of these implementations require FPGA reprogramming to obtain the reconfigurability. However, the reconfiguration requires multiple clock cycles.

A flexible constraint length Viterbi decoder architecture with the emphasis on *circuit-level* enhancements is described in [6]. The innovations described by the authors are listed below:

1. Use of radix-4 ACS units with one level of lookahead. Radix-4 ACS units process 2 stages of the trellis per iteration (The radix-4 trellis organization is shown in figure 2.14).

2. Driving the carry and sum outputs through separate logically identical logic circuits in order to reduce the ripple delay.

3. Use of *double-buffered* register files rather than dual memories for path metric storage, in order to reduce area consumption.

4. Intelligent use of constants in the branch metric generation process to reduce its complexity and area consumption .

5. Customized write and read access to the traceback memories in order to save power over conventional register-file circuits.

6. Fine-grain inclusion of supply-gating and clock-gating circuitry in order to further reduce power.

Note that in this architecture too, no specific interconnect topology has been adhered to, for connecting the ACS units.

The Viturbo architecture uses custom-built networks to interconnect the basic processing units, without trying to match the interconnect topology to the required communication patterns in channel decoding. RECFEC uses a *mesh* network to connect the basic processing units (the mesh being a regular network and hence, a popular choice for applications in general), but strangely, uses memory to implement the data transfers needed for the Viterbi algorithm (an approach also followed in another instruction-set based architecture called *FlexiTrep*, proposed in [82]). A question that can be raised here is - are there *other* interconnection topologies which can support *flexible* channel decoders more efficiently than the choices made in such architectures? Or, in other words, how useful are the topologies mirroring the actual communication pattern of the often-used channel decoders, in efficiently supporting *flexible* channel decoders? Such a question has been addressed recently in relation to flexible turbo and LDPC decoders ([61, 60, 59, 58, 62]. The *butterfly, Benes* and *de Bruijn* networks are compared for their usefulness for these decoders and it is shown that the de Bruijn network is an attractive option for the implementation of such decoders. Answering these questions for *Viterbi* decoders is the main objective of the work in this thesis.

3.2 Motivation for the approach taken in this thesis

The motivation for our work is derived from the following observations:

1. It is evident from already available results that the choice of the interconnection network connecting the ACS units of a Viterbi decoder has a great influence on the efficiency of the architecture. The main motivation for the work in this thesis is to explore the usefulness of interconnection networks which are *related* to the actual communication graph of the Viterbi decoder, for implementing *flexible* constraint length Viterbi decoders. To the best of the author's knowledge, such a study has not been undertaken due to the fact that such interconnection networks tend to occupy relatively more silicon area. However, in modern wireless devices, the power and energy consumption is a more important constraint than the silicon area

occupation. Hence, it is of interest to see whether the use of such interconnection networks allows the derivation of efficient schemes which yield significant savings in power and energy consumption. Moreover, the cost in silicon area can be offset by *reusing* the hardware resources for other algorithms in the wireless DSP unit. A fact which points to this potential for hardware reuse is that the communication pattern required for *OFDM* modulation and demodulation is similar to that required for the *Fast Fourier Transform (FFT)*, namely a *butterfly* interconnection pattern. This pattern can be shown to be similar to the de Bruijn graph, which is the interconnection pattern required for Viterbi decoding. Of course, increasing leakage currents in modern technology nodes place additional constraints on the allowable silicon area occupation as well, but this issue is not taken into account in the work in this thesis, and is in fact a direction of future work.

2. Once silicon area occupation becomes a secondary constraint, implementing an algorithm in as *parallel* a manner as possible has potential benefits in terms of power/energy consumption. Operating parallel architectures at the maximum possible operating frequency can provide high throughputs, while for moderate throughputs, a relatively low operating frequency can be used which lowers the power/energy consumption. Thus, in all the proposed architectures, a *state-parallel* approach is taken. Specifically, the architectures are capable of executing the *largest* constraint length Viterbi decoder in a completely parallel manner. For smaller constraint length decoders, an attempt is made to execute *multiple* decoders in parallel by exploiting additional sources of parallelism like the ones mentioned in section 2.5.4.

3. The architectural schemes developed are designed to be *scalable*, in order to be able to support the flexibility requirements of future mobile standards, without any additional design effort. The hardware descriptions of the architectures are also made *parameterizable* for the same purpose.

3.3 Embedding-based approach

In this study, we assume the physical architecture to consist of N processing units which are connected together in a *de Bruijn* topology (refer figure 3.7). Assuming the processing units to be numbered from 0 through $(N-1)$, the de Bruijn topology implies that every processing unit/node i is connected to two nodes - $(2*i)\%N$ and $(2*i+1)\%N$. The interconnection network is a *multi-stage* network, where the switching of data onto the links is accomplished through a set of switching elements (multiplexer-demultiplexers) and some control logic. The switching elements allow data from any of the four incoming links of a node to be routed in one of two ways -

- To any of the outgoing links *except* the link in the direction from which the data enters the node.
- To the logic *inside* the processing unit.

In addition, data at the output of the logic inside the processing unit can also be routed to any of the outgoing links.

We use this architecture to implement a *flexible* constraint length Viterbi decoder with a maximum supported constraint length of K where $N = 2^{K-1}$. For this purpose, we place an ACS unit inside each node in the network in addition to the switching logic described earlier. As can be seen from the description of the Viterbi algorithm in chapter 2, each ACS operation accepts *two* inputs. Hence, an extra multiplexer is added to the switching logic so that *two* data items can be routed in every clock cycle to the ACS unit within each node.

In figure 3.7, two copies of the *same* processing node are drawn in each row in order to conform to the conventional representation of the de Bruijn network, as done in other texts like [50] for instance. This form of depiction is also more suitable for the explanation of the routing technique described subsequently.

Note from the description of the Viterbi algorithm in the last chapter, that the communication graph for a constraint length k decoder is a de Bruijn graph of 2^{k-1} nodes. Hence, providing support for different constraint lengths on any physical network

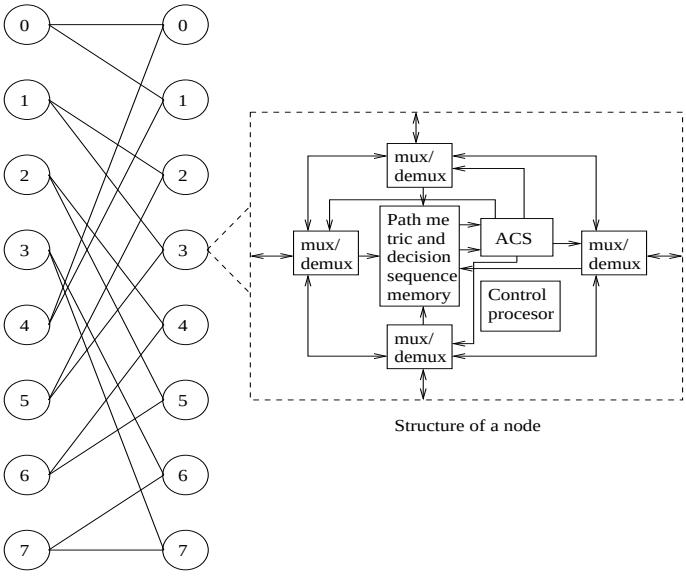


Figure 3.7: The physical architecture used for Viterbi decoder implementation

essentially means providing support for communication patterns corresponding to *de Bruijn graphs of various sizes (variable number of nodes).*

When the interconnection network is itself a de Bruijn network as in figure 3.7, this objective can be mathematically cast as the problem of *embedding* smaller de Bruijn graphs on a larger de Bruijn graph. Embedding of graphs is a well-established field in computer science, which relates to the realization of different graphs, called *guest* graphs on a particular *host* graph. In any embedding, there are four parameters which are used to ascertain the quality of the embedding, namely the *dilation*, *congestion*, *load* and *expansion* [50]. The *dilation* refers to the maximum amount by which we have to *stretch* an edge of the guest graph in order to embed it on the host graph. The *congestion* refers to the maximum number of guest graph edges embedded onto a single host graph edge. The *expansion* refers to the ratio of the number of nodes in the host graph to the number of nodes in the guest graph. The *load* refers to the number of guest graph nodes embedded onto a single node in the host graph. Quite obviously, in terms of actual hardware realizations like flexible constraint length Viterbi decoders, these parameters have a significant impact on the *quality* of the realization.

Consider a smaller de Bruijn decoder having n states to be realized on a larger physical de Bruijn interconnection network having N nodes. First, it is necessary to decide on a *mapping* of the n states onto the N nodes. We assume the most straightforward mapping, wherein the n states of the decoder are mapped onto the first n nodes of the network (node numbers 0 through $(n-1)$). Given this mapping (which implies a load of 1), we aim to obtain a method of routing the edges of the smaller *guest* graph using the links of the larger *host* graph in such a way that we obtain the *smallest* dilation and congestion. Note that one clock cycle is needed per link traversal on the physical network (a multi-hop interconnection network is assumed). This choice allows the maximum possible operating frequency to be independent of the number of nodes in the network, which aids the scalability of the architecture. While lesser dilation implies lesser latency per decoded bit, lesser congestion simplifies the logic needed for routing data on the network. In fact, a congestion of *1* is highly desirable, since no link-sharing logic would be necessary in that case.

In order to obtain a *scalable* method which is valid for any N and n, we first aim to characterize mathematically, the required connections between the states of the decoder to be realized and the connections available between the nodes on the network. In an N node de Bruijn network, each source node u is connected to two destination nodes, an *even* numbered destination node given by $(2 * u)$ *mod* (N) and an *odd* numbered destination node given by $(2 * u + 1)$ *mod* (N) (see the left side of figure 3.8). If u is represented by an N bit binary string $u = u_{\log(N)} \ldots u_1$, then the even numbered destination node has the binary representation $u_{\log(N)-1} \ldots u_1\mathbf{0}$ and the odd numbered destination node has the binary representation $u_{\log(N)-1} \ldots u_1\mathbf{1}$. (Throughout this discussion, $log()$ implies $log_2()$).

The host de Bruijn network is assumed to have *bidirectional* links (the reason will become clear as the embedding technique is explained subsequently). In that case, from every source node u, there exist links to *four* destinations, shown in column 1 of Table 3.1. Each link represents a specific bit level transformation on the *source node numbers (SNNs)* which transforms them to the *destination node numbers (DNNs)*, as indicated

Table 3.1: The set of destinations for a given source available on the physical (host) de Bruijn network and the corresponding bit level transformation of the source node which yields the destination node

Destination number	Transformation effected by the path	Modified node number
$u_{\log(N)-1} \ldots u_1\mathbf{0}$	left shift and shift in a 0	$u * 2$
$u_{\log(N)-1} \ldots u_1\mathbf{1}$	left shift and shift in a 1	$u * 2 + 1$
$\mathbf{0}u_{\log(N)} \ldots u_2$	right shift and shift in a 0	$\frac{u}{2}$
$\mathbf{1}u_{\log(N)} \ldots u_2$	right shift and shift in a 1	$\frac{u+N}{2}$

in Table 3.1.

For the purpose of illustration, refer figure 3.8. On the left side, is a 4-state decoder communication graph that is to be realized on an 8-node physical de Bruijn network shown on the right side. The physical network corresponds to a maximum supported constraint length of 4.

As shown in bold in figure 3.8, certain edges of the guest graph are *directly* available on the host graph. In general, *all the edges/paths originating from SNNs* 0 *through* $(\frac{n}{2} - 1)$ *of the guest graph are directly available on the physical network.* We call these paths the *direct* paths. This observation can be explained as follows: the guest graph paths represent the transformations i to $(2i\%n)$ or i to $(2i+1)\%n \; \forall \; i$ in SNNs 0 through $(\frac{n}{2} - 1)$. As long as $i <= (\frac{n}{2} - 1)$, the results of these operations (in other words, the DNNs) remain the same *even if* the modulus operation is performed with respect to a different number $N > n$. Since the number of physical nodes in the de Bruijn network (N) satisfies this condition, these paths are directly available on the network.

Now, consider the paths originating from SNNs $\frac{n}{2}$ through $(n - 1)$. For these paths, the results of a modulus operation performed with respect to n will *not* be the same as the results of a modulus operation performed with respect to N, when $N > n$. Consequently, these paths are not available directly on the physical network. *They have to be explicitly routed.*

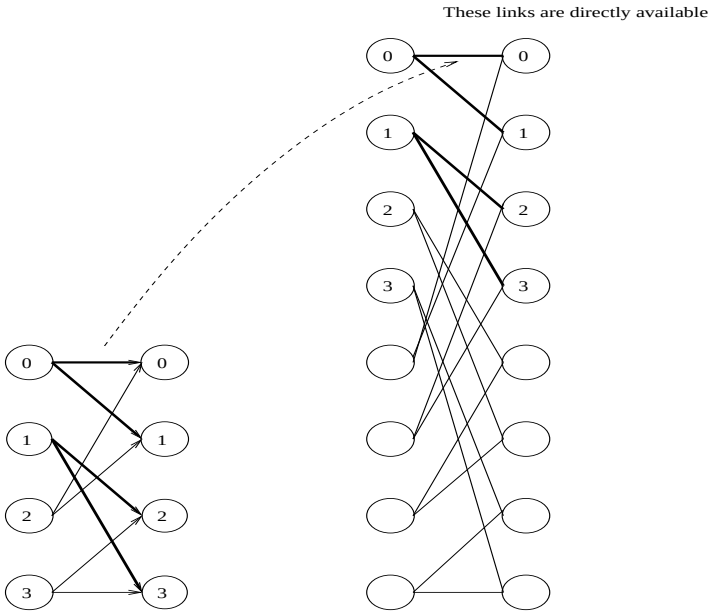


Figure 3.8: A 4-state decoder to be realized on an 8-node network. The links drawn in bold show those edges of the guest graph which are directly realizable on the physical network.

Consider the path $3 \rightarrow 2$ from the guest graph of figure 3.8 which needs to be explicitly routed on the physical network. Since the physical network contains 8 nodes, we represent all SNNs and DNNs with $log(N) = log(8) = 3$ bits. In binary representation, we need the path $011 \rightarrow 010$. In other words, the SNN 011 needs to be transformed to DNN 010 through some *sequence* of the bit-level transformations afforded by the physical network.

As seen from table 3.1, in a single step, the physical network allows a *left* or a *right* shift of a given number, with the insertion of either a 0 or a 1. We first perform the same transformation as is afforded in the guest graph - for a first step, we *left* shift the SNN once, and insert a 0 from the LSB position: $011 \rightarrow 11\mathbf{0}$ (the inserted bit is shown in bold).

Now, the DNN (2) is contained in the bit-field [1:0] of the node number. However, there is a 1 to the *left* of the DNN (called MSB_{guest}) which needs to be converted to a 0. Since a bit can only be *shifted* left or right through the physical links, this 1 needs to be shifted *out* of the node number and a 0 needs to be shifted *in*. This can be done by the addition of two more transformations (a left shift followed by a right shift) as shown

below: 011 → 110 → 10**0** → **0**10 = 2 = DNN (again, the inserted bits are shown in bold). The other required paths can also be realized by the same set of transformations.

This basic method, which begins with *left* shifts and ends with *right* shifts is called the *Left Shift First (LSF)* method.

Note that supporting both left shifts and right shifts allows us to obtain a lower step count (or dilation or latency) for performing a given transformation, since it allows us to *reuse* some of the bits of the SNN to form the DNN. If only one kind of shift were supported, the *entire* DNN would need to be shifted in from one of the ends, which would require more number of steps. For this reason, the physical links in the network of figure 3.7 are assumed to be *bidirectional*.

How many left shifts and right shifts are needed in general in the LSF method? Assume that $\frac{N}{n} = r$. Then, there will be *log(r)* 0s starting from the MSB of the node number (called MSB_{host}), in the binary representations of all the SNNs which are transformed by the LSF method. This can be brought out in another way. The node numbers are represented with binary strings which are $log(N)$ bits long, whereas the states need only $log(n)$ bits for their binary representation. Hence, in the binary representation of each state, there will $log(N) - log(n) = log(r)$ 0s starting from the MSB_{host}. The MSB_{guest} needs to be shifted out through all the $log(r)$ bit positions (which requires $log(r) + 1$ left shifts), and the flipped bit shifted back in through these $log(r)$ bit positions (which requires $log(r)$ right shifts). As an example, $log(r) = 1$ in the above example. The SNN to DNN transformation needs $log(r) + 1 = 2$ left shifts and $log(r) = 1$ right shift. The *dilation* incurred by the LSF method is $2log(r) + 1$.

Now, lets consider the possibility of the *same* physical link being traversed by two different paths routed according to the LSF method. The condition where a physical link occurs on more than one path (either at the same location or at different locations on the paths) is called a *conflict* in the following discussion (an example is provided after lemma 3.1). The following lemma defines the conditions under which a conflict occurs in the LSF method.

LEMMA 3.1. *A conflict is possible in the LSF method iff the ratio of the DNNs of two paths is a power of 2.*

Proof. During the left shifts or right shifts, we *multiply* or *divide* an intermediate node number by 2. Hence, a conflict will occur *iff* the numbers we *start* with during the left shifts, or the numbers we *end* with during the right shifts are such that their ratio is a power of 2. Now, considering only the left shifts' segment of any two paths, a conflict will occur iff *the ratio of the SNNs is a power of 2.* However, since the SNNs for which the LSF method is applied are constrained to be in the range $\frac{n}{2}$ to $(n-1)$, the ratio of two *SNNs* can never be a power of 2. Hence, there cannot be a conflict in the left shifts' segment of two paths. However, conflicts are possible in the right shifts' segment of two paths, as there are paths in any particular realization for which the ratio of the DNNs is a power of 2. □

As an example, consider the paths $5 \rightarrow 2$ and $6 \rightarrow 4$ which are present in an 8-node de Bruijn graph (ratio of DNNs $= \frac{4}{2} = 2^1$). If these were to be realized on a 32-node network, according to the LSF method, the paths would be laid out as follows:

$5 \rightarrow 2$: 00101 $\rightarrow$ 01010 $\rightarrow$ 10100 $\rightarrow$ **01000** $\rightarrow$ **00100** $\rightarrow$ 00010

$6 \rightarrow 4$: 00110 $\rightarrow$ 01100 $\rightarrow$ 11000 $\rightarrow$ 10000 $\rightarrow$ **01000** $\rightarrow$ **00100**

As shown in bold, the link $8 \rightarrow 4$ is traversed by both these paths, causing a conflict (or in embedding theoretic terms, a congestion of greater than 1). In terms of physical realization, supporting such link sharing would require extra switching logic which would add to the hardware complexity of the decoder. Also, with increasing decoder sizes, the amount of link sharing between paths may increase, thus adding a disproportionate amount of hardware complexity to the decoder. Hence, it is of interest to investigate methods by which such link sharing can be reduced, or ideally eliminated.

Is there a method to remove such conflicts in all possible cases in the LSF method? We notice that there is a freedom of choice available in terms of the bits that are *inserted* during the left shifts or right shifts, since these bits are all shifted *in* and then shifted *out*; in other words, they do not form part of the DNN. It is possible that a particular

choice of these bits helps in reducing/eliminating the conflicts in the LSF method. In the following lemma, we present one possible choice of these bits that eliminates all conflicts of the type illustrated above, alongwith its proof.

LEMMA 3.2. *All conflicts in the LSF method can be avoided if a 1 is shifted in at the second left shift on all paths.*

Proof. Consider two SNNs i and j. We trace the path laid out from these SNNs while factoring in the above mentioned modification. At the first left shift, a 0 or 1 may be shifted in, depending on whether the DNN is even or odd. This is represented by k_1 in the following expression. The left shifts' segment of the path can be represented as - $i \rightarrow (2^1.i + k_1) \rightarrow (2^2.i + 2^1.k_1 + \mathbf{1}) \rightarrow \ldots$ *(after log(r) shifts)*$\ldots 2^{log(r)}.i + 2^{log(r)-1}.k_1 + 2^{log(r)-2}$. But $2^{log(r)} = \frac{N}{n}$. Hence, we get the intermediate node number after $log(r)$ left shifts as $(\frac{N}{n}.i + \frac{N}{2n}.k_1 + \frac{N}{4n})$ (for the special case of $\frac{N}{n} = 2$, this node number will be just $(2 * i + k_1)$. At the next left shift, the MSB_{guest} is shifted out. For the particular case of $\frac{N}{n} = 2$, this step is the second left shift, when a 1 is shifted in. Hence, the node number can be represented as - $2 * (2 * i + k_1) + 1 - N$. For the case of any other value of $\frac{N}{n}$, the resulting node number can be expressed as - $2 * (\frac{N}{n}.i + \frac{N}{2n}.k_1 + \frac{N}{4n}) - N$
$= \frac{N}{n}(2.i + k_1 + \frac{1}{2} - n)$
(Note that the general expression works *even* for the case when $\frac{N}{n} = 2$. This can be attributed to the fact there are no more fractional terms in the expression for the node number after $log(r) + 1$ left shifts. We take only this general expression forward in the rest of the proof.)

Similarly, for the SNN j, we get the corresponding node number as - $\frac{N}{n}(2.j + k_2 + \frac{1}{2} - n)$ (k_2 is used instead of k_1).

Now, consider the ratio of the node numbers on the two paths at the *start* of the right shifts - $\frac{\frac{N}{n}(2.i + k_1 + \frac{1}{2} - n)}{\frac{N}{n}(2.j + k_2 + \frac{1}{2} - n)}$. This evaluates to $\frac{2(2.i + k_1 - n) + 1}{2(2.j + k_2 - n) + 1}$. This is clearly a ratio of*odd* numbers, which can never be equal to a power of 2. Hence, due to the modification proposed above, the numbers at the start of the right shifts segment of any two paths are always such that their ratio is *not* a power of 2.

Now, we track the behaviour of this ratio as the numerator and denominator are both divided by 2, due to the right shifts. Note that for all left shifts *after* the second left shift, 0s are shifted in. There are $log(r) + 1 - 2 = log(r) - 1$ such 0s. As such, for the first $log(r) - 1$ *right* shifts, a *perfect division* (without a remainder) of the numerator and denominator is performed as these 0s are shifted out. At the *last* right shift, the 1 shifted in at the second left shift is shifted out. Hence, an *imperfect* division (division of an odd number by 2) of the numerator and denominator is performed.

All the perfect divisions represent a *reduction* of the fraction representing the ratio of the numbers at the start of the right shifts' segments of the two paths. Hence, if the numbers at the start of the reduction are such that their ratio is not a power of 2, this property will be retained at all successive reduction steps. This means that there is no combination of a node a on the first path and node b on the second path, such that $\frac{a}{b}$ is a power of 2. As this is a necessary condition for the same link to be traversed on both the paths in the right shifts' segment, a conflict will never occur between the two paths in the right shifts' segment. Of course, this property may not be retained at the last right shift, since imperfect division does not imply a *correct* reduction of the fraction. But this is of no consequence, as no more shifts are performed after this point.

Hence, there will be no conflict in the right shift's segment of any two paths. As it is already demonstrated that there is no conflict possible in the left shifts' segment of any two paths (refer lemma 3.1), *all* conflicts between any two paths laid out by the LSF method can be avoided with the above modification. Note that conflicts between the *left* shifts' segment of one path and the *right* shifts' segment of another path need not be investigated as the physical network is assumed to have bidirectional links.

□

As an illustration, we retrace the two paths $5 \rightarrow 2$ and $6 \rightarrow 4$ with the above modification: (the second inserted bit is shown in bold)

$5 \rightarrow 2$: 00101 $\rightarrow$ 01010 $\rightarrow$ 1010**1**$\rightarrow$ 01010 $\rightarrow$ 00101 $\rightarrow$ 00010

$6 \rightarrow 4$: 00110 $\rightarrow$ 01100 $\rightarrow$ 1100**1**$\rightarrow$ 10010 $\rightarrow$ 01001 $\rightarrow$ 00100

As can be seen, there is no common link between the two paths. Thus, the conflict is avoided.

One can see that the path latency in the LSF method will go on increasing as the $r = \frac{N}{n}$ ratio goes on increasing. In other words, this method will not be very efficient when relatively smaller decoders have to realized on large-sized physical networks. The efficiency can be raised if the MSB_{guest} which is to be modified is shifted out through the *LSB* of the node number (called LSB_{host}) - in other words, shifted out from the *right*. This leads to another method of realization of the paths called the *Right Shift First (RSF)* method, which is *complementary* to the LSF method, in that it begins with right shifts and ends with left shifts.

As an example, if the path $3 \rightarrow 2$ of an $n = 4$-state decoder has to be realized on an $N = 64$-node physical network, it can be done as follows:

$3 \rightarrow 2$: $000011 \rightarrow 000001 \rightarrow 000000 \rightarrow 000001 \rightarrow 000010$

The latency in this method, which is 4, is lesser than the latency of 9 that would be incurred if the path had been laid out by the LSF method as shown below:

$3 \rightarrow 2$: $000011 \rightarrow 000110 \rightarrow 001101 \rightarrow 011010 \rightarrow 110100 \rightarrow 101000 \rightarrow 010100 \rightarrow 001010 \rightarrow 000101 \rightarrow 000010$

In the path laid out by the RSF method above, note that the there is a redundant link sequence - $000001 \rightarrow 000000 \rightarrow 000001$. In other words, the MSB_{guest} of the SNN is the *same* as the MSB_{guest} of the DNN, and ideally this bit should not be shifted out and shifted in again. Upon closer observation, we see that there can be more than one bit which is common between the SNN and DNN. For instance, consider the SNN-DNN pairs $14 \rightarrow 12$ and $14 \rightarrow 13$ of a 16-state decoder. The first *two* bits of 14(1110) and 12(1100) are the same. In general, we observe that for an SNN having a run of a 1s including the MSB_{guest}, the DNNs have a run of $(a-1)$ 1s including the MSB_{guest}. This can also be explained in another manner: in an n-node de Bruijn graph, an SNN i is connected to DNNs $(2i)\%n$ and $(2i+1)\%n$. Note that the RSF method is applied only to SNNs $\frac{n}{2}$ through $(n-1)$, where each SNN has an MSB_{guest} of 1. Due to the multiplication by 2 specified by the de Bruijn graph, this MSB_{guest} is shifted *out* of the

node number representation, reducing the length of the run of 1s from the MSB_{guest} by one.

Due to the presence of such a commonality between the SNN and DNN, we need to shift out only the last $(log(n) - (a-1))$ bits of the SNN and shift in the respective bits of the DNN.

Now, we need to decide the number of shifts to be performed for each SNN-DNN pair, and hence, we need to characterize the set of SNNs that have the *same* run-length of 1s including the MSB_{guest}. This can be done as follows.

Consider again the example of a 16-state decoder. The first *range* of SNNs with a run-length of one 1 including the MSB_{guest} exists from 8 to 11 (i.e $\frac{n}{2}$ to $\frac{3}{4}n - 1$). The second range of SNNs with a run-length of 2 exists from 12 to 13 (i.e. $\frac{3}{4}n$ to $\frac{7}{8}n - 1$). In general, the range m of SNNs, where each SNN has a run-length of m 1s including the MSB_{guest} exists from $\frac{(2^m-1)n}{2^m}$ to $\frac{(2^{m+1}-1)n}{2^{m+1}} - 1$, where m takes values from 1 to $log(n)$. Thus, there exist $log(n)$ ranges for an n-state decoder.

For all paths in a range m, we perform $(log(n) - (m-1))$ right shifts to remove the last $(log(n) - (m-1))$ bits of the SNN and then an equal number of left shifts to shift in the last $(log(n) - (m-1))$ bits of the DNN.

Note that the *same* data is to be routed to both the even and odd DNNs from a given SNN. Hence, we route the path to the odd DNN *along* the even DNN path, except at the last step, where we shift in a 0 on the even DNN path and a 1 on the odd DNN path.

As an example, consider the realization of a 16-state decoder on a 256-node network. The paths $14 \rightarrow 12$ and $14 \rightarrow 13$, which are associated with the SNN range 3, will be laid out with the help of $log(n) - (m-1) = 4 - (3-1) = 2$ right shifts and 2 left shifts as follows:

$14 \rightarrow 12 : 00001110 \rightarrow 00000111 \rightarrow 00000011 \rightarrow 00000110 \rightarrow 00001100.$

$14 \rightarrow 13 : 00001110 \rightarrow 00000111 \rightarrow 00000011 \rightarrow 00000110 \rightarrow 00001101.$

In this method, the path latency is *different* for different ranges of SNNs, but the same for all paths originating from SNNs within a range. For all paths originating from SNNs in a range m, the path latency is $(2log(n) - 2(m-1))$. Hence, the maximum path latency occurs for paths in range 1, and it is $2log(n)$.

In order to obtain the minimum possible dilation for a given realization, we decide between the use of the LSF and RSF method by comparing the maximum path latencies. In other words, if $2log(r) + 1 >= 2log(n)$, we use RSF to lay out all the paths, else we use LSF to lay out all the paths.

In the RSF method too, conflicts can occur if the method shown above is applied directly. Consider the paths 10→4 and 11→6 which are required in an $n = 16$-state Viterbi decoder, realized on an $N = 256$-node physical network.

10→4: 0000101$\underline{0}$ → **00000101** → **00000010** → ...

11→6: 0000101$\underline{1}$ → **00000101** → **00000010** → ...

The conflict occurs as soon as the last bit that is *different* between the two SNNs (shown underlined) is shifted out. To avoid such conflicts, the *differentiating* bits must somehow be *retained* in the node number during the shifting process. Analogous to the case with the LSF method, the bits which are inserted during the right shifts and shifted out during the left shifts in the RSF method do not form part of the DNN, and hence we can choose a particular sequence of these bits which leads to the elimination of such conflicts.

Note that for two paths to be non-conflicting, the following needs to hold true - for any link L occurring at step s on path 1, the same link should not occur on path 2 *either at step 's' or at any other step.* To prove this, it is necessary to prove that a node pair $a \rightarrow b$ occurring on path 1 in one direction of shifting does not occur on path 2 *at any step* along the segment with the same direction of shifting. Although this is the general condition, lemma 3.3 proves that in case of the RSF method, it is enough to prove that no node pair $a \rightarrow b$ occurring on a path 1 at step s can occur *at the same step* on a path 2 (of course, in the same direction of shifting), in order to prove that path 1 and path 2

are non-conflicting paths. This relaxation of proof conditions is used in the subsequent lemmas, (3.4, 3.5 and 3.6). These lemmas describe one sequence of inserted bits which can successfully eliminate all conflicts, alongwith the proof for its effectiveness.

LEMMA 3.3. *A node occurring at a step s on a path laid out by the RSF method can never occur at a different step $s' \neq s$ on any other path laid out by the same method.*

Proof. In the RSF method, for an SNN in a range m, $log(n) - (m-1)$ right shifts followed by an equal number of left shifts are performed. The RSF method is applied under the condition that $2log(r) + 1 >= 2log(n)$, which translates to $log(r) >= log(n)$. This means, that the number of right shifts or left shifts performed is always *lesser* than or *equal* to $log(r)$.

One of the ways to prove the exclusivity of node numbers occurring on any two paths is to show the existence of a difference in a particular *bit-field* within the binary representations of the node numbers on the two paths. This is the approach taken in this proof and the subsequent proofs related to the RSF method.

We first introduce the modification needed in the RSF method in order to eliminate all potential conflicts - during the right shifts, we propose to shift in the *complement of the even DNN* corresponding to a given SNN, *LSB first*.

For example, the path 10→4 required in the realization of a 16-state decoder on a 256-node network will be realized with the above modification as follows:

10→4:

Even DNN = 0100

Even DNN complement = 1011

Path: 00001010 → **1**0000101 → **11**000010 → **011**00001 → **1011**0000 → **011**00000 → **11**000001 → **1**0000010 → 00000100

Keeping this modification in mind, note that the *first* bit shifted in during the right shifts (referred to as B henceforth) according to the modified RSF method is always 1, because it is the complement of the LSB of the even DNN (which is a 0).

The bit-field referred to in this proof is formed by the concatenation of B, the $log(r)$ leading 0s and the m 1s present in the binary representation of the node number at any intermediate step in the RSF method. For instance, consider the realization of a 16-state Viterbi decoder on a 256-node network. The bit-field referred to is shown in bold in the following representation of an example node 10 (range 1) in binary form (shown after the first right shift): **00001**010 $\rightarrow$ **10000**101.

Note that this bit-field *moves* within the binary representation of the node number as the right shifts and left shifts are performed. Now, consider an intermediate node number on *another* path and at a *different* step number $s^{'}$. In general, we can suppose that there will be some *partial overlap* between the referred bit-fields in the two node numbers. Consider the extent of the partial overlap, or more appropriately, the extent of the *offset* between the locations of the two bit-fields. Since the number of right shifts or left shifts is *less than or equal to* $log(r)$ as proved earlier in the proof, there will be a *maximum offset* of *less than or equal to* $log(r)$ bits between the locations of the above-mentioned bit-fields in the node numbers on the two paths. Since the length of the referred bit-field is $log(r) + 2$ bits, such an offset will always cause one of the $log(r)$ leading 0s to be present in the node number on one of the paths, at the bit position corresponding to either B or MSB_{guest} of the other node number (note that B and MSB_{guest} are both 1). Due to this difference, we can say that a particular intermediate node number occurring at step s on one path will never occur on another path at a *different* location $s^{'}$.

□

Keeping lemma 3.3 in mind, it is enough to prove that an intermediate node number occurring at a step s on one path will not occur on another path at the *same* step s, in order to prove that two paths laid out by the RSF method are non-conflicting. This property is made use of in lemma 3.4 and lemma 3.5.

LEMMA 3.4. *All potential conflicts between any two paths laid out by the RSF method in a given range 'm' are avoided if the complement of the even DNN (corresponding to each SNN) is shifted in, MSB first, during the right shifts, for all paths.*

Proof. (refer figure 3.10 for an example).
Consider two paths whose SNNs are such that, the pth bit is the last bit that is different (or complementary), counting from the LSB of the node number (LSB_{host}). Let SNN1 have the binary representation $a_{log(n)} \dots a_1$ and SNN2 have the binary representation $b_{log(n)} \dots b_1$.

Let DNN1 have the binary representation $c_{log(n)} \dots c_1$ and DNN2 have the binary representation $d_{log(n)} \dots d_1$ (we do not specify whether the DNNs are even or odd since that does not affect our proof). In accordance with *left* shift of the SNN defined by the de Bruijn topology, the $(p+1)$th bit (c_{p+1} and d_{p+1}) will be the last bit that is different (counting from LSB_{host}), between the DNNs.

For the proof, we consider the right shifts' segment and left shifts' segment separately. In both sections, we strive to demonstrate that an intermediate node number occurring on one of the paths at a step l does not occur on the other path at the *same* step. This is enough to prove that path 1 and path 2 are non-conflicting, keeping lemma 3.3 in mind.

Right shifts' segment:
For the first $(p-1)$ right shifts, a_p and b_p are present in the node representations on the two paths at each shift, thus making the intermediate nodes on the two paths different. Hence, the two paths cannot conflict in this part. At the next right shift, a_p and b_p are shifted out from LSB_{host}. In the worst case, when *no* other bits are different between SNN1 and SNN2, the same node number will be reached on both paths. However, even in this case, the same *link* will not be traversed as the bits shifted *in* at the next right shift, $c_{(p+1)}$ and $d_{(p+1)}$, are complements of each other. Also, these two bits remain part of the node representations on the two paths all through the remaining right shifts (at the last right shift, they are $((log(n)-(m-1))-(p+1))$ positions from MSB_{host}), and hence the two paths will continue to be distinct throughout the remaining right shifts.

Left shifts' segment:
During the left shifts, for the first $((log(n)-(m-1))-(p+1))$ left shifts, $c_{(p+1)}$ and $d_{(p+1)}$ remain part of the node representations and hence, the two paths continue to be distinct. At the next shift, these bits will be shifted out from MSB_{host}, but a_p and b_p are

shifted in from LSB_{host}, which remain part of the node representation for all remaining left shifts. Hence, the node numbers will continue to be different for all the left shifts too. □

For instance, the conflict in the RSF method indicated above is removed by using the above modification as follows:

10→4: Even DNN = 0100 Even DNN complement = 1011 Path: 00001010 → **1**0000101 → **11**000010 → **011**00001 → **1011**0000 → **011**00000 → **11**000001 → **1**0000010 → 00000100

11→6: Even DNN = 0110 Even DNN complement = 1001

Path: 00001011 → **1**0000101 → **01**000010 → **001**00001 → **1001**0000 → **001**00000 → **01**000001 → **1**0000011 → 00000110

LEMMA 3.5. *There can be no conflict between two paths belonging to different ranges, when laid out by the RSF method. .*

Proof. (Refer the example given after this lemma.)
We can assume that $m1 < m2$, since the proof will recursively hold to cover all the ranges. Let the path in range 1 be denoted by path 1 and the path in range 2 be denoted by path 2. Again, as in the previous lemma, we aim to prove that a node number occurring at a step s on one of the paths will not occur at the *same* step on the other path. This is enough to prove that the two paths are non-conflicting, keeping lemma 3.3 in mind.

In the right shifts' segment, the node representation at any intermediate step s on path 2 has a substring containing $log(r)$ zeroes followed by $m2$ 1s, whereas the *same* substring in the node representation at the same step s on path 2, ('same' here indicates the substring occupying the same bit positions), contains $log(r)$ zeroes followed by $m1$ 1s followed by $(m2 - m1)$ 0s.

In all the left shifts, the node representation at any intermediate step s on path 2 has a substring containing $log(r)$ zeroes followed by $(m2 - 1)$ 1s, whereas the same field in the node representation at the same intermediate step s on path 1 has a substring containing $log(r)$ zeroes followed by $(m1 - 1)$ 1s and $(m2 - m1)$ 0s.

It follows from the observations made above that as long as $m1 \neq m2$, a node number occurring in left shift (right shift) section of path1 at a step s will not occur at the same step in the left shift (right shift) section of path2. The case of the same link occurring in the *left* shift section of path 1 and the *right* shift section of path 2, or vice versa need not be considered, since the physical network is assumed to have bidirectional links. □

This can be illustrated by the following example: consider the realization of the paths 10→4 and 13→10 (belonging to ranges 1 and 2 respectively) of a 16-state decoder realized on a 256-node physical network. The bit-fields referred to in the above lemma are shown in bold face in the paths traced out below.

10→4: Even DNN = 0100 Even DNN complement = 1011 Path: **00001**010 → 1**0000**10**1** → 11**000010** → 011**00001** → 10110000 → 011**00000** → 11**000001** → 1**0000**010 → **00000**100

13→10: Even DNN = 1010 Even DNN complement = 0101 Path: **00001**101 → 1**0000**11**0** → 01**000011** → 101**00001** → 01**000010** → 1**0000**101 → **00001**010

It is seen that there is no link common to both the paths.

The following lemma formally ties up the above two lemmas.

LEMMA 3.6. *There can be no conflict between any two paths laid out by the modified RSF method.*

Proof. A potential conflict can be of two types: a conflict between two paths in the same range m or a conflict between two paths in two different ranges $m1$ and $m2$. Hence, the proof follows from lemma 3.4 and 3.5. □

The next two lemmas prove that neither the LSF method nor the RSF method causes any conflict with any of the paths originating from SNNs 0 through $(\frac{n}{2} - 1)$.

LEMMA 3.7. *There can be no conflict between a path laid out by the LSF method and any of the direct paths.*

Proof. One of the nodes on each link of the direct connections is always less than $\frac{n}{2}$. Also, the direct links perform a *left* shift of the node number. Hence, for a conflict to occur, an intermediate node number less than $\frac{n}{2}$ should occur in the *left* shifts' segment of a path laid out by LSF. However, in the LSF method, the left shifts' segment starts

with a number greater than or equal to $\frac{n}{2}$ and the node number is multiplied by 2 with each successive left shift. Hence, in this segment, there can be no conflict with the direct connections. □

LEMMA 3.8. *There can be no conflict between a path laid out by the RSF method and any of the direct paths.*

Proof. In the RSF method, for the first $(log(n) - (m-1))$ steps, we perform right shifts. Due to the assumption of bidirectional physical links, there cannot be a conflict in this section. Note that the first bit to be shifted in during the right shifts is always a 1 as explained earlier in lemma 3.3. At the end of the right shifts, this 1 is located at the position $log(N) - (log(n) - (m-1) - 1) = (log(r) + m)$. As shown in lemma 3.3, in the cases where the RSF method is applied, the condition $log(r) \geq log(n)$ holds true. As such, we can also say that $(log(r)+m) \geq (log(n)+1)$. Note that the MSB_{guest} corresponds to the bit position $log(n) - 1$. Hence, the 1 that is shifted in at the first left shift is always located to the *left* of the MSB_{guest}, at the end of the right shifts. This means the node number at the end of the right shifts will always be greater than or equal to $\frac{n}{2}$. Now, at each left shift, this node number is multiplied by 2, with or without the addition of a 1. Hence, the intermediate node numbers in the left shift section will always be greater than $\frac{n}{2}$. Hence, there can be no conflict by the same argument as that given in lemma 3.7. □

The last lemma formally ties up all the above lemmas in order to prove that the presented methods alongwith the associated modifications eliminate all possible conflicts. (in other words, they ensure a congestion of 1).

THEOREM 3.9. *There is no conflict between any of the paths laid out by the proposed realization technique.*

Proof. Any potential conflict can be classified into one of the following types:

1. a conflict between two paths laid out by the LSF method
2. a conflict between two paths laid out by the RSF method
3. a conflict between a path laid out by the LSF method and a direct connection.

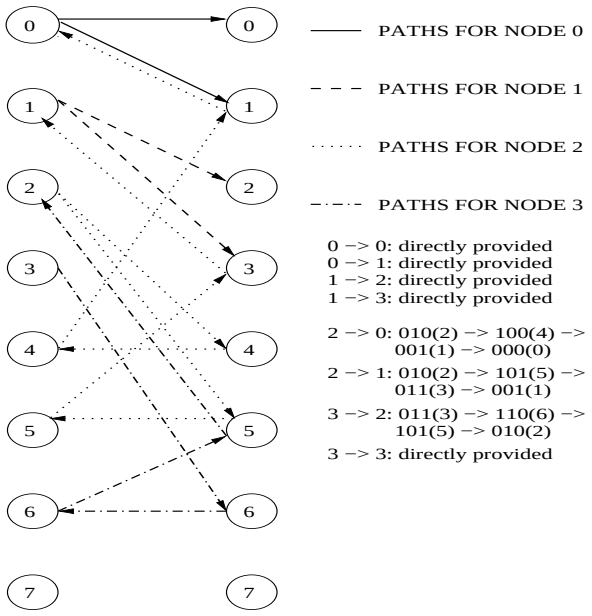


Figure 3.9: Realization of a 4-state decoder on an 8-node de Bruijn network (LSF method)

4. a conflict between a path laid out by the RSF method and a direct connection.

Hence, the proof follows from lemmas 3.2, 3.6, 3.7 and 3.8. □

The LSF method is illustrated in figure 3.9 while the RSF method is illustrated in figure 3.10.

In terms of hardware, each of the switching elements shown in figure 1 is controlled by settings stored in a memory. The contents of the memory are calculated *offline* in accordance with the technique described above and addressed by the input indicating the current constraint length. Note that changing the constraint length takes only one clock cycle.

The *average number of cycles per decoded bit* indicates the throughput that can be expected from the architecture. In this method, the average number of cycles per decoded bit is *different* for different constraint lengths. We can assume that the ACS operation is performed in one clock cycle. Now, when the LSF method is used, the average number of cycles per decoded bit will be $2log(r)+1$, and when the RSF method is used, this number will be $2log(n)$. Hence, if N is a perfect square, the maximum value of the average number of cycles per decoded bit ($= log(N)$) will occur for the decoder

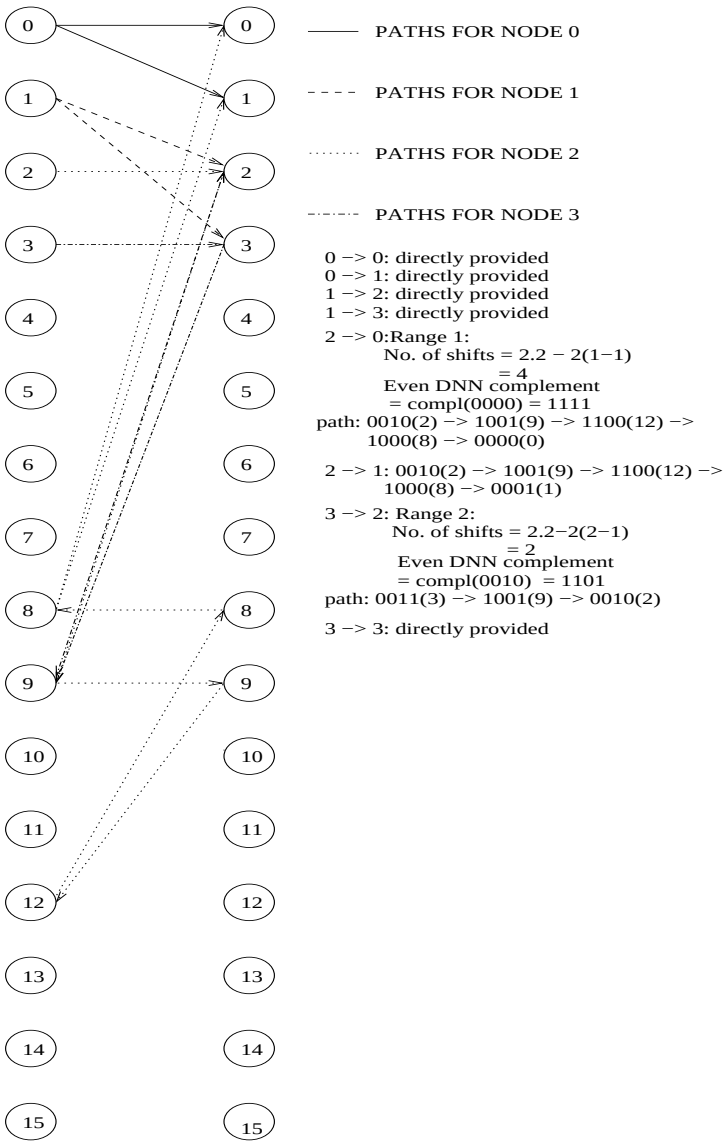


Figure 3.10: Realization of a 4-state decoder on a 16-node de Bruijn network (RSF method)

with $n = \sqrt{N}$ states, when $log(n) = log(r) = \frac{log(N)}{2}$. If N is not a perfect square, the maximum latency ($= log(N)$) will be incurred by the decoder with $n = \sqrt{(2 * N)}$ states, when $log(n) = \frac{(log(N)+1)}{2}$.

In order to obtain increased throughput, one might think of the possibility of pipelining the transport of metrics through the physical links. This would allow metrics to be inserted into the network *every* clock cycle, which would eliminate the need for stalling the ACS units in between successive stages. However, two properties of the architecture make this possibility infeasible. Firstly, the lengths of different paths realized by the RSF method are *different*. For ensuring correct functionality alongwith pipelining, *buffer* registers are required on all the paths, in order to equalize all the path lengths. Moreover, additional logic is needed to read off data from the *appropriate* buffer register according to different constraint lengths. This, combined with the fact that the bitwidths of the physical links are relatively large in keeping with the register transfer method (64 bits for a constraint length 9 decoder), makes the support for pipelining disproportionately costly in terms of area and also power.

The implementation results of this and all succeeding architectures are plotted in figures 5.1 and 5.2 of chapter 5.

Viewed in an embedding-theoretic way, the described technique proves the result that there is *at least one* way to *embed* smaller de Bruijn graphs on a large de Bruijn graph with *bidirectional* links, incurring a maximum latency of $log(N)$ while also eliminating any congestion.

3.4 Variable state-to-node assignment-based approach

The usefulness of the embedding theoretic result derived above towards Viterbi decoding is *limited* by the fact that the technique does not provide a way to execute *multiple* decoders on the network in the case of smaller constraint lengths. The various switch settings or configuration bits also require storage which leads to disproportionate silicon area consumption as detailed in chapter 5. These observations suggest the need to search

for a more efficient scheme for realization of Viterbi decoders on a de Bruijn network.

It is possible to recast the embedding problem as one of embedding *multiple* smaller de Bruijn graphs onto a larger de Bruijn graph and search for mathematical manipulations similar to the ones presented in the above discussion to minimize the dilation and congestion. However, the embedding approach in general has one potential disadvantage - the *static* mapping of the decoder states onto the nodes. A straightforward mapping of state i onto node i was assumed in the previous section in order set up a definite environment for mathematical analysis. It is quite obvious that the method of mapping of the states onto the nodes will have a great effect on the quality of the mapping in general. Moreover, from a hardware perspective, it is not even necessary to have a static mapping of decoder states onto the physical nodes. Intuitively, allowing the same state to be mapped to different nodes at different time instants can potentially lead to a more efficient way of realizing Viterbi decoders on the network. This suggests that it is necessary to move away from embedding-based approaches and explore other kinds of schemes which also utilize the available freedom of assigning the same state computation to different nodes at different time instants. In what follows, one such improved scheme is described.

An example of this scheme for a 4-state decoder implemented on an 8-node network is shown in figure 3.11.

As shown, the scheme starts with a scheduling where state i is mapped to node $\frac{N}{n} * i$, where n is the number of states. From this point, the results are just sent along the existing links at each stage, and subsequent states are scheduled depending on where the appropriate metrics are available. After $log(n)$ stages, the metrics are all located in the top n nodes. From here, it is not possible to bring any two metrics to a common node for the next stage of ACS operations by only following the existing links. Hence, a certain number of *metric rearrangement stages* are performed, in which the path metrics are transported along *specific* links to bring them back to the initial position, from where the entire scheme repeats. These specific links can be identified by considering the node where each metric is located after $log(n)$ operations and the node where the metric needs

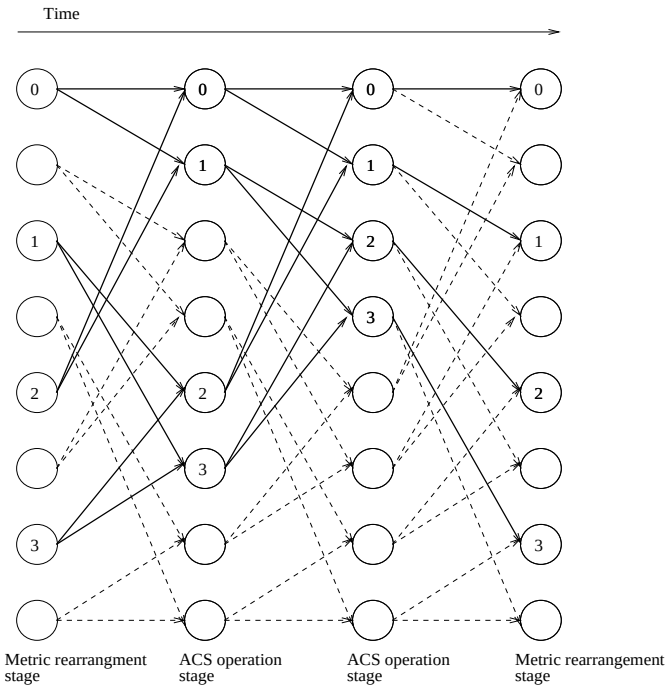


Figure 3.11: State scheduling scheme for a 4-state decoder realized on an 8-node de Bruijn. The nodes represent ACS units. The links along which path metrics are transferred are shown as normal arrows, while the unused links are shown as dotted arrows.

to reach in order for the scheme to begin again.

Consider, for example, the metric at the third node (node number 2) at the penultimate stage in figure 3.11, which contains the updated metric of state 2. This metric needs to be routed to node 4 for the scheme to begin again. We can get from node number 2 to node number 4 by performing one *left shift*. In general, we observe that the metric at any node i can be transferred back to the starting point by performing $log(r)$ *left circular shifts* where $r = \frac{N}{n}$ as before.

In terms of the network, the data is transferred over the links which connect node i to $(2i)\%(N)$. Note however, that the same node number transformation can also be performed by *right circular shifts*. More precisely, the transformation achieved by performing $log(r)$ left circular shifts can also be achieved by performing $log(n)$ right circular shifts, since $log(n) + log(r) = log(N)$. The particular kind of shift chosen for a given constraint length can be decided based on which of these two values is lesser, in order to minimize the number of metric rearrangement stages. The right circular shifts

require a smaller number of steps when $n <= \sqrt{N}$. Additional logic is however needed to perform the appropriate transfer of metrics after the $log(N)$ operations.

In this method, different metrics may be transferred over the same physical link during the metric rearrangement stages (note that this does not happen during the ACS computation stages). However, there will never be a case when the two metrics have to be transported over the same physical link *in the same clock cycle*. This is ensured due to the use of *circular* shifts during the metric rearrangement stages. As such, no extra logic is necessary in order to contain any congestion of the physical links. Hence, no effort is made to transfer only one data item through each physical link, as was done in the embedding approach in 3.3.

This architecture is not entirely time-efficient in the sense that the ACS units are not active during the metric rearrangement stages. However, the number of cycles for which the ACS units have to be stalled is restricted to $(min(log(n), log(r)) + 1)$ (for instance, in figure 3.11, there are $min(log(r), log(n)) + 1 = 1 + 1 = 2$ metric rearrangement stages). The average number of cycles per stage can be calculated as the number of cycles needed for one iteration of the scheme divided by the number of stages in one iteration of the scheme, which is $\frac{log(n)+min[log(n),log(r)]+1}{log(n)}$. The *maximum* value of this expression occurs for the *minimum* constraint length to be supported. This can be intuitively explained as follows - for larger constraint lengths, the penalty for the metric rearrangement is spread over a larger number of ACS operation stages, since $log(n)$ is relatively large. If we consider the minimum constraint length to be supported as 3, which corresponds to a 4-state decoder, then the maximum average number of cycles per stage is limited to $\frac{(log(4)+log(4)+1)}{log(4)} = 2.5$ (assuming that $log(n) < log(r)$ for $n = 4$, which would be the case when the constraint lengths specified in modern standards are to be supported). It is important to note that this is *constant* irrespective of the size of the decoder ie. irrespective of the maximum constraint length to be supported. Thus, the performance of this architecture scales well with decoder size. Another important point to be noted is that the scheme just described requires the links to have only *half-duplex* capacity (since data may be transferred in one of two directions over a physical link but never

simultaneously in both directions), whereas the embedding-based technique described in section 3.3 requires the links to be bidirectional. Hence, this architecture is expected to be relatively more area-efficient.

For implementation purposes, an expression for calculating the state which is scheduled on a node in a given clock cycle is required. This expression can be found as follows:

The clock cycle numbering is begun from the last intermediate metric rearrangement stage (the leftmost stage in figure 3.11). Thus this stage corresponds to clock cycle number 0 and the ACS operation stages are cycle numbers 1 and 2. Now, from figure 3.11, we can see that it is possible to group the states of the decoder into *sets* which are mapped onto *consecutive* nodes. For instance, in cycle 0, there are 4 sets of states, each containing 1 state. In cycle 2, there are 2 sets, each containing 2 states and so on. In general, the number of states in each of the sets in cycle number s is 2^s. The *gap* (in terms of number of nodes between the starting points) between two successive sets is 2 in cycle number 0 ($= \frac{N}{n}$ in general) , 4 in cycle number 1 ($= 2 * \frac{N}{n}$ in general) and so on. If the stage number is s, this gap can be expressed by $(2^s * \frac{N}{n})$. s in the equation goes from 0 to $log(n)$.

The aim here is to obtain an expression for the *node* on which a particular state is scheduled in a particular clock cycle. In cycle number s, each set contains 2^s states which are scheduled consecutively. This is expressed by having $i\%2^s$ as one term in the expression, where i is the state whose mapping is to be found. Another *additive* term is needed in order to express the gap between the sets. Note that this gap should be applied to states which are separated by 2^s, as can be seen from figure 3.11. Hence, this additive term takes the form - $2^s * \frac{N}{n} * quotient(\frac{i}{2^s})$. Hence, the overall expression can be presented as follows:

In stage s, decoder state i is mapped to physical node z where - $z = 2^s * \frac{N}{n} * quotient(\frac{i}{2^s})$ + $i\%2^s$.

The significant advantage in this scheme over the embedding-theoretic scheme is that it is possible to easily extend this scheme to support the execution of several smaller

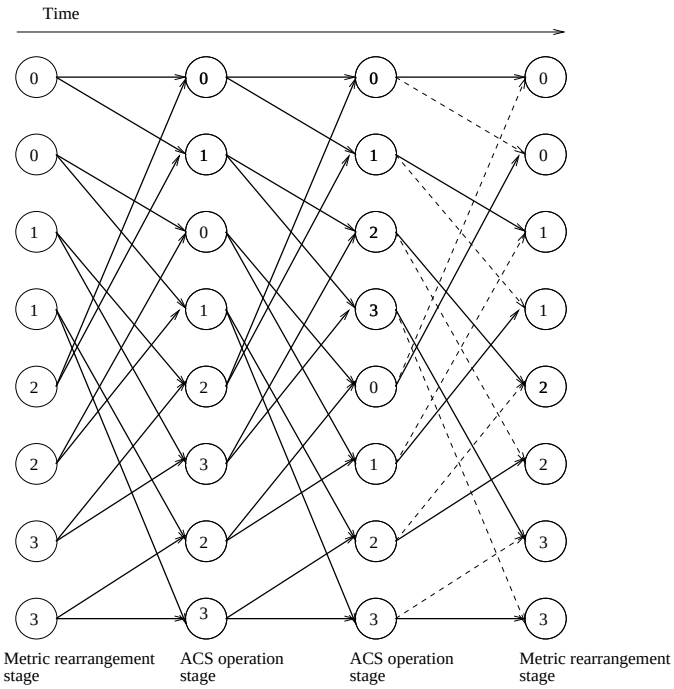


Figure 3.12: Two 4 -state decoders parallelized on the 8-node network. The nodes present ACS units. The links along which path metrics are transferred are shown as normal arrows, while the unused links are shown as dotted arrows.

trellises *in parallel* on the physical network. In particular, $\frac{N}{n}$ trellises can be executed in parallel on the network as shown in figure 3.12. **(For instance, *four* trellises of an $n = 64$-state decoder can be supported on an $N = 256$-node architecture)**. In this case, state i of trellis j is initially mapped to node $\frac{N}{n} * i + j$. Otherwise, the scheme is similar to the one-trellis case. To provide data for the parallel trellises, any of the methods discussed in the previous chapter can be used.

In this case too, during the metric rearrangement cycles, the metric at node i is transferred to the initial point through $log(r)$ left circular shifts, or an equivalent number of right circular shifts. For the parallel trellises case, the expression for finding the state scheduled at a node is found to be as follows: At node z, during stage s, the scheduled state is given by - $(z\%2^{s+1}) + 2^{s+1} * quotient(\frac{z}{2^{s+1} * \frac{N}{n}})$. This expression is based on analysis similar to that presented for the unparallelized case.

There is also one more advantage of this architecture with respect to the embedding-theory-based architecture, in addition to the ability to exploit parallelism. The physical links required in the embedding theoretic scheme need to be *bidirectional* or *full-duplex*,

since data may be transported in both directions over a physical link simultaneously. On the other hand, in the scheme described in this section, data is always transported in *one* of two directions over a physical link in any given clock cycle. This can be observed from the fact that the data movement in this scheme in any given clock cycle can be represented *either* as left shifts or as right shifts - it is never required to represent the data movement as a *combination* of left and right shifts. As a result of this difference, the silicon area consumption of this architecture can be expected to be considerably lesser than that of the embedding theory-based architecture. As seen in chapter 5, due to these two advantages over the embedding theory-based architecture, the variable state-to-node assignment-based architecture turns out to be much more area-efficient than the former.

This scheme is the main contribution of this thesis. The implementation results and inferences are provided in figures 5.1 and 5.2 of chapter 5.

Chapter 4

Alternative architectures

In the last chapter, two flexible constraint length Viterbi decoder architectures based on the *de Bruijn* interconnection network were described. In this chapter, three more architectures based on the *shuffle-exchange*, *flattened butterfly* and *2-D mesh* respectively are described. The shuffle-exchange and flattened butterfly networks belong to the same family of networks as the de Bruijn network. However, they provide a different tradeoff between area occupation and throughput, and hence, a comparison between the relative *area-efficiencies* of these networks is of interest. The 2-D mesh is a popular interconnection network, which topology-wise, occupies relatively lesser area compared to the de Bruijn, shuffle-exchange and flattened butterfly networks. As such, it is also of interest to compare the area-efficiencies of a mesh-based Viterbi decoder architecture with the other architectures.

4.1 Implementation on a shuffle exchange network

An N-node shuffle exchange graph has edges defined as follows ([50]):
There is an edge between node i and j if:

- j is a single left or a right circular shift of i. (This edge is called a shuffle edge.)
- j and i are different only in the LSB. (This edge is called an exchange edge.)

Figure 4.1 shows a physical realization of an 8-node shuffle exchange graph, where the edges are shown to be bidirectional links (allowing transfer of data in both directions simultaneously).

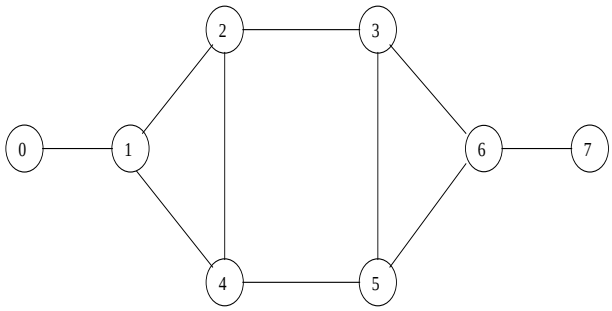


Figure 4.1: 8-node shuffle exchange network. The links are bidirectional.

Note that *all* the links in a de Bruijn network can be described as shuffle links whereas the shuffle-exchange network has a mix of shuffle and exchange links. As exchange links connect consecutively numbered nodes, while shuffle links connect nodes which are relatively farther away in terms of their numbers, we can say that the shuffle-exchange network contains more near-neighbour communication than the de Bruijn network. As such, the shuffle exchange network is expected to occupy relatively less silicon area, compared to the de Bruijn network. [40] shows that an N-node de Bruijn graph can be embedded in an N-node shuffle exchange network with a dilation of 2 and a congestion of 1. This is shown for an 8-node case in figure 4.2. The path from each SNN to one of the DNNs is laid along a *shuffle* link, whereas the path to the other DNN is laid along a shuffle link followed by an *exchange* link. The connectivity pattern for SNNs in the first half (0 to $\frac{N}{2} - 1$) is different from the connectivity pattern for SNNs in the latter half. For SNNs in the first half, the path to the *even* DNN is laid out using a shuffle link, whereas the path to the *odd* DNN is laid out using a shuffle link followed by an exchange link. As an example, the path $1 \rightarrow 2$ is realized in one hop directly ($1 \rightarrow 2$), while the path $1 \rightarrow 3$ is routed as: $1 \rightarrow 2 \rightarrow 3$. For the latter half of the SNNs, the path to the *odd* DNN is laid out using a shuffle link, while the path to the *even* DNN is laid out using a shuffle link followed by an exchange link. As an example, the path $5 \rightarrow 3$ is available directly, whereas the path $5 \rightarrow 2$ is realized as $5 \rightarrow 3 \rightarrow 2$.

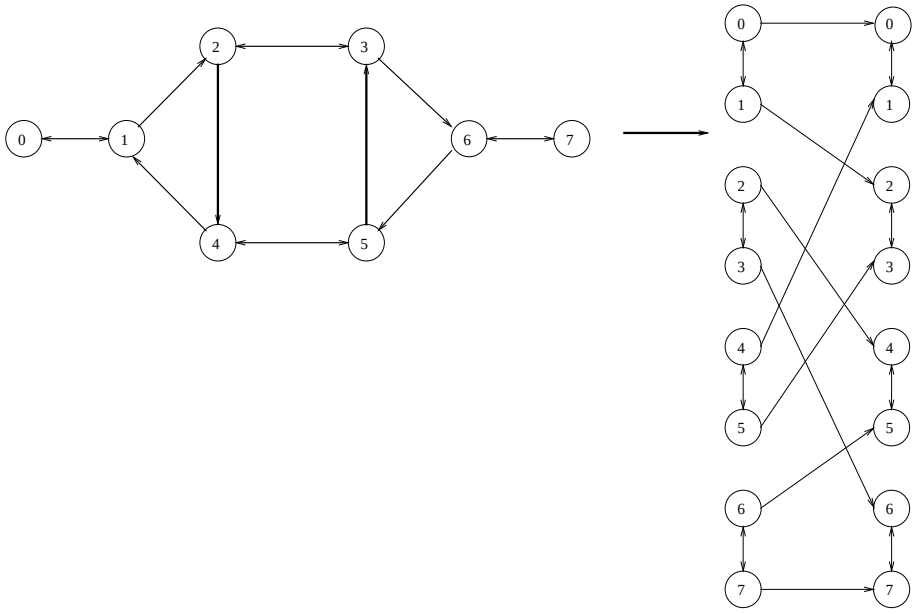


Figure 4.2: 8-node de Bruijn network realized on the 8-node shuffle exchange network. The links are shown with appropriate directions. An alternative representation which is similar to the representation of the de Bruijn network is shown on the right hand side.

The realization of the de Bruijn network on the shuffle-exchange network is represented in a different manner on the right hand side of figure 4.2. This is more similar to the representation of the de Bruijn network in figure 3.7 in chapter 3, and is more suitable for the depiction of the realization of the Viterbi decoder on the network.

As shown in figure 4.2, *all* the links of an N-node shuffle exchange network do not need to be bidirectional in order to realize an N-node de Bruijn network. More precisely, only the *exchange* links need to be bidirectional, whereas the shuffle links are unidirectional.

With the directivity of the links defined as in figure 4.2, it is possible to *reuse* the scheme described for the de Bruijn network in section 3.4 of chapter 3, with only some restructuring of the switching logic. Figure 4.3 shows the variable state-to-node assignment scheme executed on a shuffle-exchange network, for an 8-state decoder.

As mentioned before, one of the two paths originating from each SNN has a latency of 2 clock cycles. Hence, each of the $log(n)$ ACS operation stages requires 2 clock cycles. In the metric rearrangement stages, all the stages *except* the last metric rearrangement stage require data to be routed only along the shuffle links. As these links are directly

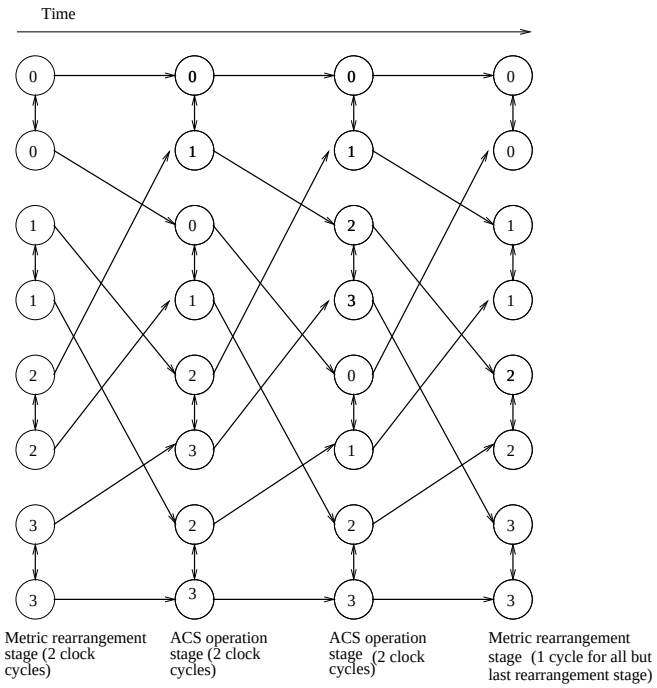


Figure 4.3: Variable state-to-node assignment scheme executed on a shuffle-exchange network. 2 4-state decoders are shown parallelized on an 8-node network

available in the physical networks, these stages require only one clock cycle each. However, in the last metric rearrangement stage, data is required to be transferred along the communication pattern described by the de Bruijn graph. Hence, the last metric rearrangement stage requires two clock cycles.

Now, the average number of cycles needed per stage can be calculated as - (no. of cycles needed for $log(n)$ ACS stages + no. of cycles needed for metric rearrangement stages)/($log(n)$). This is equivalent to - $\frac{2*log(n)+(min[log(n),log(r)]+1)+1}{log(n)}$. The expression for calculating the state to be scheduled on a particular node remains the same as in case of the de Bruijn implementation.

As in the case of the de Bruijn-based architecture, the maximum value of the average number of cycles per stage is obtained for the smallest constraint length to be supported. If we assume the smallest constraint length to be supported to be 3 (4-state decoder), then the maximum value of the average number of cycles per stage is - $\frac{2*log(4)+log(4)+1+1}{log(4)} = 4$ (assuming that $log(n) < log(r)$ at $n = 4$, which would be the case when the constraint lengths specified in modern standards are to be supported). This is again *constant* with

respect to the decoder size. Hence it can be said that even this architecture scales relatively well with decoder size, performance-wise.

Compared to the de Bruijn network-based architecture executing the same scheme, (presented in section 3.4), the shuffle-exchange network-based architecture has a higher value of the average number of clock cycles per stage. This means, that the shuffle-exchange network-based architecture will not provide as much throughput as the de Bruijn network-based architecture. However, it is expected to occupy *lesser silicon area* than the de Bruijn network, as explained earlier, although this area reduction will be slightly *offset* by the fact that the exchange links in the shuffle-exchange network-based architecture are required to be *bidirectional* or *full-duplex*. As such, it is interesting to see if the area reduction of the shuffle-exchange network-based implementation achieved with respect to the de Bruijn network-based implementation is *greater* or *lesser* than the reduction in the average performance. The comparison is detailed in chapter 5.

4.2 Implementation on a flattened butterfly network

The butterfly graph is another graph closely related to the de Bruijn and shuffle exchange graphs. [50] contains detailed material on the properties of the graph. The definition of the butterfly graph is reproduced here from [50] in order to aid comprehension. The r-dimensional butterfly has $(r+1) * 2_r$ nodes and $r * 2_{r+1}$ edges. The nodes correspond to pairs w, i, where i is the *level* or *dimension* of the node ($0 \leqslant i \leqslant r$) and w is an r-bit binary number that denotes the *row* of the node. Two nodes w, i and $w`, i`$ are linked by an edge if and only if $i` = i + 1$ and either:

- w and $w`$ are identical
- w and $w`$ differ in precisely the ith bit.

An 3-dimensional butterfly (with $(3+1) * 2_3 = 32$) nodes is shown in figure 4.4.

In order to execute the Viterbi algorithm on this network, the nodes are replaced with ACS units and the edges are replaced with *bidirectional* links. Now, it is necessary

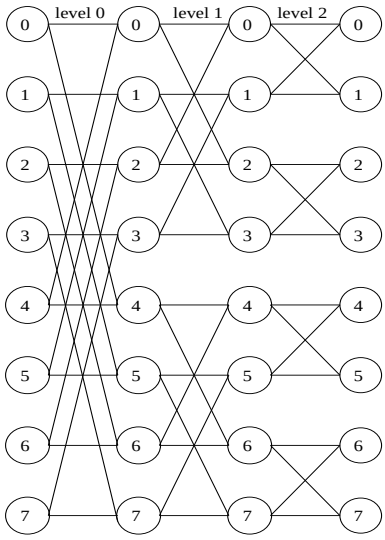


Figure 4.4: A 3-dimensional butterfly. Level i *straight* edges link nodes in the same row for $0 \leqslant i \leqslant r$. Level i cross edges link nodes in rows that differ in the ith bit (here the numbering of the bit positions starts from the MSB of the node number).

to find a state-scheduling scheme which extracts the highest possible efficiency out of this architecture. It turns out that a scheme already proposed in prior literature is able to fit this requirement.

This scheme is proposed in [39]. It basically associates the communication pattern required for a Viterbi decoder of N states, with the communication pattern required for a *bitonic sort* of N items. The communication pattern for a bitonic sort of $N = 8$ items is provided in figure 4.5 as an example.

For adapting this communication pattern to Viterbi decoders, a modification is done to the normal operation of the ACS units. Specifically, each ACS unit *retains a copy* of its result with itself, while also sending the result out of the node at every stage.

Refer figure 4.6, where the state scheduling scheme for executing a 8-state Viterbi decoder on a 16-item bitonic sorting network is shown.

To explain the operation, we trace one set of ACS operations across stages. The ACS computations of states 0 and 8 of the decoder are scheduled on nodes 0 and 8 in stage 0. After computing the results, each node retains a copy of the result as well as exchanges it result with the other node. Thus, after the completion of stage 0, both nodes 0 and

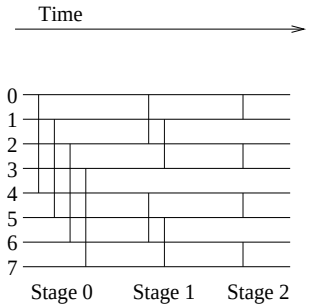


Figure 4.5: The communication pattern needed for the bitonic sort of 8 items. Each link represents a *comparator* and connects the two data items which are compared and sorted by the comparator at a given time instant. Each comparator sorts its two inputs in ascending order, putting out the *lower* valued input on the upper output line and the *upper* valued input on the lower output line.

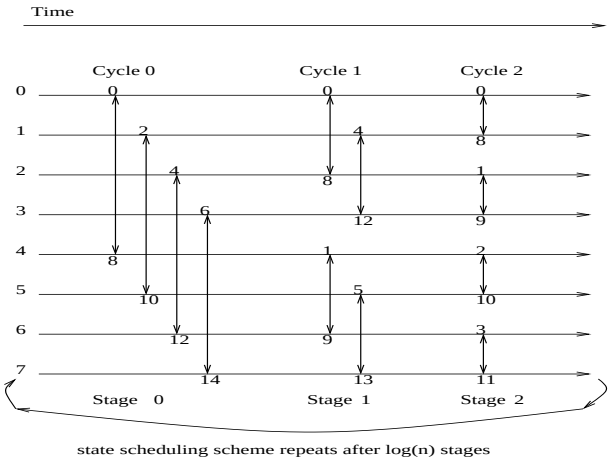


Figure 4.6: An 8-state Viterbi decoder implemented with the communication pattern akin to an 8-item bitonic sorting network. The double-arrows links represent the exchange of results between two ACS units. The numbers beside the double-arrows indicate the states whose ACS operations are scheduled on the respective nodes in a given stage.

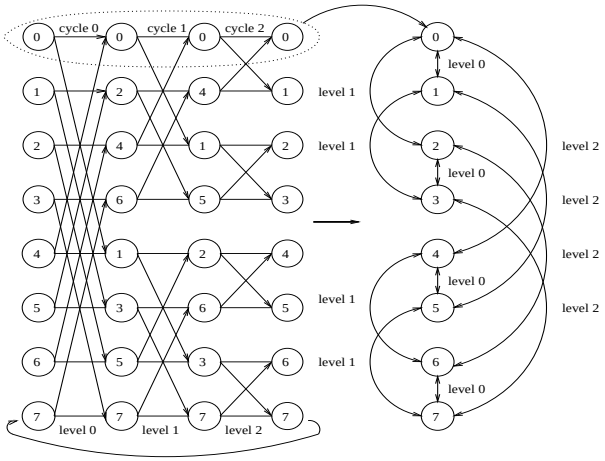


Figure 4.7: 8-node decoder realized on the 8-node butterfly network. The figure on the right shows the equivalent flattened butterfly network. The state assignment scheme repeats every log(n) cycles; in this case it repeats every log(8)=3 cycles.

8 contain the results (updated metrics and decision sequences) of states 0 and 8. Now, according to the connection pattern defined by figure 2.3 in chapter 2, these results are needed for the ACS computations of states 0 and 1 of the next stage (stage 1). Hence, in the next stage, the ACS computations of states 0 and 1 are scheduled on nodes 0 and 8 respectively. This process repeats for all other state-pairs of the decoder, in each stage. After $log(n)$ stages, the state-to-node assignment goes back to the normal binary order at the start of the scheme, and thus, the entire scheme gets repeated again and again. Stage i gets executed in clock cycle $i\%(log(n))$.

The reason for the use of this scheme is the fact that an $(log(N))$-butterfly network *directly* fits the communication pattern for a bitonic sort of N items. This is shown in 4.7 for the 8-state decoder case.

Hence, by using the technique of [39] alongwith this network, we can obtain the *completely* time-efficient version of the flexible constraint length Viterbi decoder, on which the average number of cycles per stage is 1 irrespective of the constraint length or decoder size (assuming of course, that successive received symbols are transported to the appropriate level of ACS units as and when required). The scheme is shown for an 8-state decoder realized on an 8-node butterfly network in the left hand side of figure 4.7. The number inside each node shows the state whose ACS computation is scheduled

on that node in a particular clock cycle. Initially, state i of the decoder is scheduled on node i of the network. The state to be scheduled on a node in subsequent clock cycles can be obtained by performing a *left circular shift* of the state number scheduled on that node in the previous cycle.

If the ACS units are *duplicated* to form the levels of the butterfly network, then additional parallelism needs to be exploited to keep all the ACS units busy. This parallelism could be achieved by using any of the techniques described in section 2.5.4. However, one can see that this network has a considerably higher degree compared to the de Bruijn and shuffle-exchange networks (a 256-node butterfly network will have a degree of 8, whereas the 256-node de Bruijn and shuffle-exchange networks will have a degree of only 4). As such, this network is considerably more *complex* when compared to the de Bruijn and shuffle-exchange networks and is expected to occupy a relatively large silicon area. This limits the *scalability* of the network. In order to enhance the scalability, we constrain the capabilities of the network by placing only *one* node for each *row* of the butterfly. This leads to a different network called the *2-ary 4-flat flattened butterfly*, described in [21]. This transformation is also shown in figure 4.7. In this architecture, each of the $log(n)$ stages of the scheme described above is executed using the *same* processing nodes in successive cycles. Level i links are used for exchange of results in cycle $i\%N$. Additional logic is incorporated into the processing units in order to implement the required routing.

Since an N-node butterfly network contains $\frac{N}{n}$ instances of any smaller n-node butterfly network, this architecture is also easily parallelizable as shown in figure 4.8. In this case, the scheme is repeated after only $log(4) = 2$ stages and level 2 links are not used at all. Additional logic is incorporated into the processing units in order to support the dynamic variation of constraint length.

Note that the butterfly topology is inherently more interconnection-intensive compared to the de Bruijn and shuffle-exchange topologies. Also, as mentioned earlier, the links in the flattened butterfly are required to be *bidirectional* or *full-duplex* in order to be able to execute each stage of the described scheme in *1* cycle. Hence, this architecture is expected to occupy considerably more area with respect to the de Bruijn and

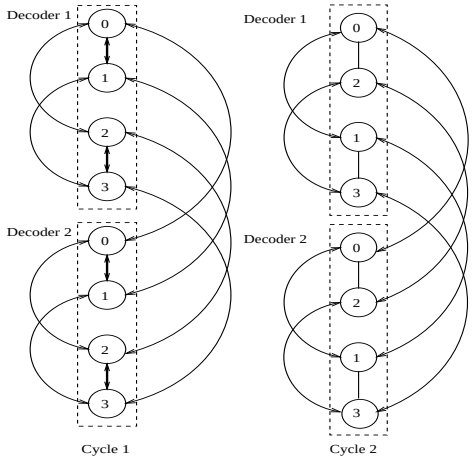


Figure 4.8: Two 4-state decoders realized on the 8-node flattened butterfly network. The active links in each cycle are shown in bold.

shuffle-exchange network-based architectures. However, as can be seen from the values of the average number of cycles per decoding stage, this architecture potentially provides much higher throughputs compared to the de Bruijn and shuffle-exchange network-based architectures. As such, it is interesting to *compare* the increase in area of the flattened butterfly architecture over the de Bruijn and shuffle exchange architectures with the increase in average performance. Chapter 5 details the comparison.

4.3 Implementation on a mesh network

It is interesting to see how the architectures described earlier compare with architectures based on *small* wire-area interconnection topologies in terms of area efficiency. For this purpose, we choose to compare these architectures with a flexible constraint length Viterbi decoder implemented on a *two-dimensional mesh* network. The 2D-mesh network is a widely used interconnection network which is known to be compact, since the links provide only local neighbour communication, similar to the exchange links in the shuffle-exchange network (see section 4.1).

The 2D-mesh is different from the other networks considered in this thesis, in that there is no fixed rule for numbering the nodes in the network. Hence, in the most general sense, it is necessary to search the entire search space of node numberings and

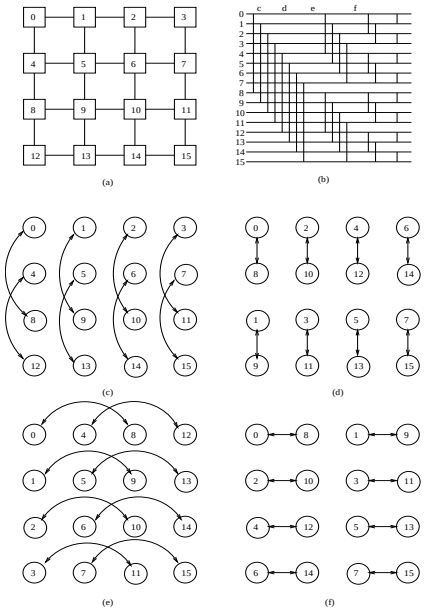


Figure 4.9: (a)16-node mesh network, numbered in row major order. (b)The trellis communication pattern drawn considering the location of the metrics - it resembles a 16-node bitonic sorting network. (c)-(f) The communication between the nodes in each of the log(n) = log(16) = 4 stages.

corresponding routing strategies in order to identify the optimal scheme for executing the convolutional/Viterbi decoder on the mesh. Obtaining a *scalable* method, which works for arbitrary decoder sizes becomes difficult in that case, since the results of the search would, in general, be different for different decoder sizes. As such, we use a method proposed in [39], for Viterbi decoder implementation on a mesh, which is scalable, while still being reasonably efficient in terms of performance.

This method/scheme is just the bitonic sorting-based scheme described in 4.2, executed on the 2D-mesh. The scheme is illustrated in figure 4.9, for a 16-state decoder. The bitonic-sorting-based scheme is explained earlier in section 4.2. A brief recapitulation is given below.

Analogous to the normal binary numbering of nodes at cycle 0 in figure 4.7, in this architecture, the scheme starts with a *row-major* numbering of all the nodes. This numbering indicates the location of the path metrics of the 16 states at cycle 0. With reference to figure 2.3 of section 2.5 in chapter 3, the basic communication pattern in

the execution of the Viterbi algorithm can be seen as a butterfly, with two producer nodes sourcing data to two consumer nodes. Now, according to the technique described in [39], in each stage of the trellis computation, the two nodes corresponding to the two producer states within every butterfly *exchange* the results of their respective ACS operations. Each node also *retains* a copy of the result of the current ACS operation within itself. The data required for the two consumer states are available at both the nodes, and thus in the next stage, the two consumer states are scheduled on these two nodes.

For example, consider the nodes 0 and 8 of the original mesh (sub-figure (a) of figure 4.9). In cycle (c) shown in figure 4.9, node 0 performs the ACS operation for state 0 of the decoder while node 8 performs the ACS operation for state 8 of the decoder. Subsequently, each of these nodes retains a copy of its current result and also exchanges it with the result produced by the other node, as shown by the communication pattern in subfigure (c) of figure 4.9. Hence, at the end of first stage, each of the nodes 0 and 8 contains the updated metrics for states 0 and 8 of the decoder. Now, for a 16-state decoder, the updated metrics corresponding to states 0 and 8 are needed for the ACS operations of states 0 and 1 of the next stage. Thus, in subfigure (d) of figure 4.9, state 0 is scheduled for computation on node 0 and state 1 is scheduled for computation on node 1 (this assignment could also be reversed without losing correctness). Other state assignments are also done in a similar manner.

The communication pattern among the nodes resulting from this scheme is similar in general to that of an N-item bitonic sorting algorithm, where N is the number of nodes in the network, or equivalently the number of states in the decoder. The entire routing scheme repeats after $log(N)$ stages. As mentioned in section 4.2, the state to be scheduled in a particular clock cycle on a node is obtained by performing a left circular shift of the state scheduled on that node in the previous clock cycle.

Since the mesh interconnection topology does not directly fit the bitonic sorting communication pattern, some of the stages require multiple clock cycles for the rearrangement of data. As shown in figure 4.9, the required communication pattern maps

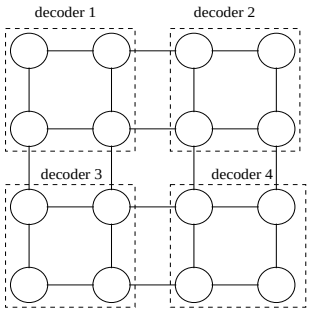


Figure 4.10: Placement of 4 4-state decoders on a 16-node mesh network.

to *row* or *column* permutations of the data within the mesh network. For a mesh implementation with m rows and n columns, there are $log(m)$ row permutations followed by $log(n)$ column permutations in each iteration of the state scheduling scheme. In addition, row permutation i requires $2^{(log(m)-1-i)}$ cycles, while column permutation i required $2^{(log(n)-1-i)}$ cycles (assuming the numbering of the row and column permutations to start from 0). Hence, the total number of cycles needed to complete one iteration of the state scheduling scheme is $\sum_{i=0}^{(log(m)-1)} 2^{(log(m)-1-i)} + \sum_{i=0}^{(log(n)-1)} 2^{(log(n)-1-i)}$. Since $log(m) + log(n) = log(m \times n) = log(N)$ stages of the Viterbi decoder are computed in each iteration of the scheme, the average number of clock cycles *per stage* of this architecture is given as - $\frac{\sum_{i=0}^{(log(m)-1)} 2^{(log(m)-1-i)} + \sum_{i=0}^{(log(n)-1)} 2^{(log(n)-1-i)}}{log(N)}$.

Note that an n-state decoder needs a mesh of $\sqrt{n} \times \sqrt{n}$ nodes for its execution. Now, the N-node mesh network is trivially decomposable into $\frac{N}{n}$ smaller meshes, each of size $\sqrt{n} \times \sqrt{n}$. Hence, by incorporating additional control, $\frac{N}{n}$ instances of smaller n-state decoders can be realized *in parallel* on this network. This fact can be used to extend the above architecture to support multiple constraint lengths as well as to support parallelism for smaller constraint length decoders. For instance, figure 4.10 shows how 4 4-state decoders can be placed on the 16-node mesh network. Note that this scheme requires the links to be *full-duplex*.

In this architecture, the physical links are required to be *bidirectional* or *full-duplex* in order to support the described scheme. This fact *offsets* to some extent, the expected area reduction of this topology with respect to the topologies discussed earlier. Also, as can be seen from the expression for the average number of cycles per decoding stage, the

expected throughput from the mesh-based architecture is *lesser* than the throughputs expected from the other architectures. As such, it is again interesting to compare the potential area reduction obtained by the use of this architecture with the reduction in throughput. The implementation results of this architecture are presented in chapter 5.

Chapter 5

Comparison of the presented architectures

5.1 Synthesis results and inferences

The architectures of chapters 3 and 4 were described in Bluespec System Verilog (BSV) (www.bluespec.com), which is a high level hardware description language, and verified for their functionality. It is possible to obtain the *Verilog* descriptions of the architectures automatically from the BSV descriptions. These descriptions were then synthesized using Synopsys' Design Vision using 90nm Faraday libraries, for a maximum constraint length of 9, which corresponds to a physical network with 256 nodes. The maximum possible operating frequency was determined and the corresponding throughputs, silicon area consumption and power consumption were noted. The results are plotted as a graph in figure 5.1. The power consumption figures provided by Synposys Design Vision are known to be approximate since the actual activity factors are not taken into account, but they provide an indication of the expected power consumption and thus, the relative suitability of the discussed architectures in terms of this metric.

The hardware needed to supply the parallel decoders with data has not been taken into account in these realizations, since it is common to all the architectures and thus, does not affect the comparisons. Also, it is expected to be considerably lesser than the

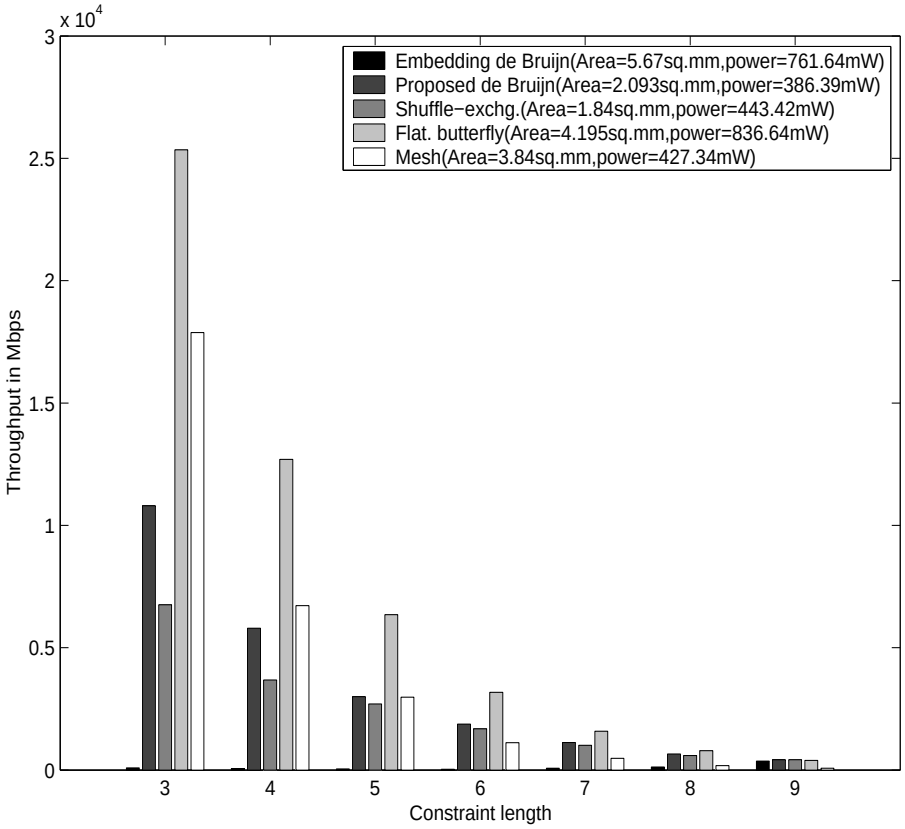


Figure 5.1: Plot showing the throughputs for different constraint lengths for the architectures considered in this paper.

area occupied by the actual interconnection network, especially for the decoder sizes considered in this paper.

For comparing these architectures, area-time products are useful metrics [39]. The "time" here refers to the average amount of time required for each output bit to be produced. To obtain this time, the average number of clock cycles per stage for each architecture *across* all supported constraint lengths is obtained and then normalized with respect to the maximum operating frequency. **For comparison in the moderate throughput region, the** $(area) * (time)$ **product is used, while comparison in the high throughput region is performed through the** $(area) * (time^2)$ **product.** These products are plotted in figure 5.2 for the architectures considered in this paper.

Note that the *lower* the $(area) * (time)$ or $(area) - (time)^2$ product of an architecture, the *more* area-efficient it is.

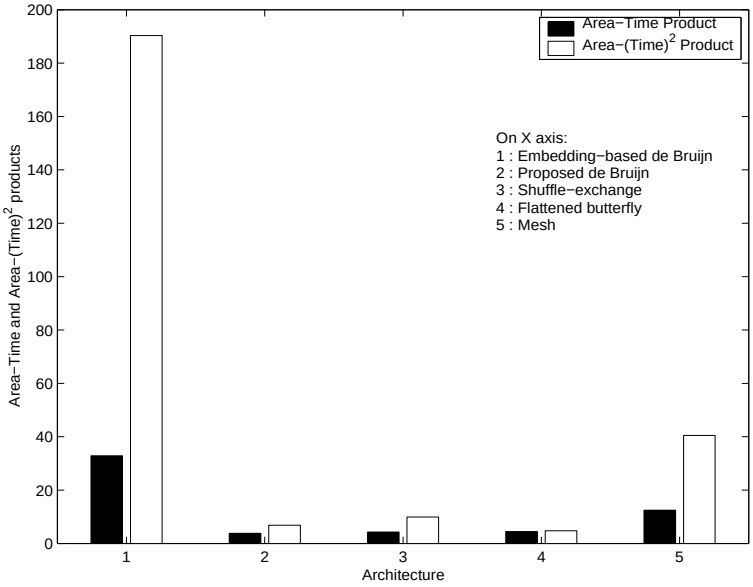


Figure 5.2: Plot showing the $(area) * (time)$ and $(area) * (time^2)$ products of all the architectures considered in this paper.

Inferences from synthesis:

1. The embedding-based architecture described in chapter 3 is seen to be the *least* area-efficient compared to all the architectures discussed in this thesis. This is due to the large amount of control information that needs to be stored in order to realize the embedding scheme and the inability to exploit the inherent parallelism present in the decoding process. Also, the links in this architecture are required to be *full-duplex*, while the links in the de Bruijn and shuffle-exchange-based architectures are *half-duplex*. This is an important factor leading to the reduction in the silicon area occupation of the de Bruijn and shuffle-exchange-based architectures.

2. The mesh-based architecture also requires full duplex links. Hence, despite the *topology* having a nominally lower silicon area requirement, the architecture actually occupies *more* silicon area than the architectures based on large-wire area topologies which require only half-duplex links. This architecture is also less area-efficient when compared to the architectures based on the large wire-area networks.

In fact, the mesh architecture does not compare favourably even with the flattened butterfly-based architecture, which also has full duplex links. The area reduction of the mesh-based architecture with respect to the flattened butterfly-based architecture is only around 8%, while the average performance drops by 115%. The less than expected area reduction in case of the mesh-based architecture compared to the flattened butterfly architecture can be attributed to the fact that the individual nodes in the mesh architecture are more complex than the nodes in the flattened butterfly-based architecture, and hence consume more area. However, the mesh implementation is far more area-efficient than the embedding-based architecture described in chapter 3, for reasons enumerated earlier in inference 1 and also on account of the topology.

For the particular case when the maximum supported constraint length is 9, the de Bruijn-, shuffle-exchange- and flattened butterfly- network-based architectures compare as follows:

3. The shuffle-exchange network-based architecture occupies 12% less area than the de Bruijn network-based architecture, but the associated performance degradation is 28%.

4. The flattened butterfly network-based architecture provides 65% more performance compared to the de Bruijn network-based architecture and 96% more performance compared to the shuffle-exchange network-based architecture, but it area occupation is 100% more than that of the de Bruijn network-based architecture and 128% more than that of the shuffle-exchange network-based architecture.

5. From figure 5.2, we see that for a maximum supported constraint length of 9, the *variable state-to-node assignment* scheme implemented on the *de Bruijn* network-based architecture (described in chapter 3) is the most area-efficient architecture in the *moderate* throughput region, while the *flattened butterfly network-based* described in chapter 4 is the most area-efficient in the *high* throughput region.

Table 5.1: Estimated energy dissipation for the synthesized architectures

Architecture	Energy dissipation per decoded bit
Embedding-based de Bruijn	10.137 nJ
Variable state-to-node assignment based de Bruijn	74.95 pJ
Shuffle-exchange network based	135.69 pJ
Flattened butterfly based	117.13 pJ
2D-mesh based	182.9 pJ

The architectures could also be compared on the basis of their *energy* dissipation per decoded bit. Table 5.1 indicates the estimated energy dissipation per decoded bit for the five architectures considered. The estimates are obtained as follows:

1. First, the average number of clock cycles needed per decoded bit for the decoder is calculated. This is taken as the average of the number of clock cycles per decoded bit over all the constraint lengths supported.

2. The available parallelism also needs to be incorporated. For this, the average number of decoded bits produced per stage is obtained across constraint lengths, and the value for the average number of cycles per decoded bit obtained in the earlier step is divided by this value.

3. This is multiplied by the time span of a single clock cycle, in order to obtain the average *time* per decoded bit.

4. This is then multiplied by the estimated power dissipation provided by Synopsys Design Vision, in order to arrive at the average *energy* dissipated per decoded bit.

Again, the de Bruijn network-based architecture with the variable state-to-node assignment scheme and the flattened butterfly network-based architecture emerge as the most attractive architectures. Note that this advantage will be more pronounced as the range of constraint lengths to be supported increases. The throughputs of these architectures do not degrade with increasing constraint lengths, whereas those of the embedding theoretic-based and 2D-mesh-based architectures degrade with increasing constraint lengths.

Chapter 6

Conclusions and future work

This chapter draws the final conclusions from the work presented in this thesis and provides some directions for future work in relation to the topics dealt with in the thesis.

6.1 Conclusions

The synthesis results in chapter 4 pertain to a maximum supported constraint length of 9. Is it possible to predict, from these results, the behavior of the area-efficiencies of the considered architectures as the decoder size increases?

It turns out that the average number of cycles per decoding stage is the parameter which yields decisive information about the area-efficiencies of the architectures:

1. For the de Bruijn network-based architecture in section 3.3 and the 2D-mesh-based architecture, the value of the average number of cycles per decoding stage is more than that for the other three architectures (described in sections 3.4, 4.1 and 4.2), even for a maximum supported constraint length of 9. Also, for these two architectures, this value *increases* with decoder size, as can be derived from the explanation of the architectures given in chapters 3 and 4. Finally, the silicon area occupation of these two architectures is greater than that of the other three architectures, even for a maximum supported constraint length of 9, and it can be expected to remain greater even for higher constraint lengths, due to the requirement of

bidirectional links. As such, it can be said with certainty that these architectures are not favourable from an area-efficiency point of view, when compared with the architectures described in sections 3.4, 4.1 and 4.2.

Now we consider the relative area-efficiencies of the variable state-to-node assignment scheme of section 3.4, the shuffle-exchange-based architecture of section 4.1 and the flattened butterfly-based architecture of section 4.2.

2. Note that the average number of cycles per decoding stage is *constant* for all these three architectures. As such, the obtained performances of the three architectures will continue to have the same proportion with respect to each other, as the decoder size increases. However, with increasing decoder sizes, the silicon area of the flattened butterfly network-based architecture will increase at a higher rate compared to the de Bruijn- and shuffle-exchange network-based architecture, due both to the higher degree/network complexity and the need for each link to be bidirectional. As such, although the flattened butterfly-based architecture is found to the most area-efficient for a maximum constraint length of 9, it can be said that this architecture will lose this optimality for higher decoder sizes.

3. Between the de Bruijn-based architecture of section 3.4 and the shuffle-exchange-based architecture, the de Bruijn-based architecture is found to be more area-efficient for a maximum supported constraint length of 9. However, the area occupation of this architecture will increase at a higher rate with increasing decoder size, compared to the shuffle-exchange-based architecture, due to the higher degree. In addition, the relative performances of these architectures will continue to maintain the same proportion due to a *constant* value for the average number of cycles per stage. As such, it can be said that for higher constraint lengths, the shuffle-exchange network-based architecture will be the most area-efficient option, compared to the other architectures considered in this thesis.

Some more general conclusions can also be drawn from the work in this thesis:

- Interconnection networks with nominally higher silicon area requirement, but which reflect the actual communication patterns required in a given application more closely, are *interesting* options for realizing the applications, especially given the capabilities of today's fabrication technology. Effort should be directed towards the development of efficient execution schemes for various functionalities on architectures based on these networks.

- For wireless digital signal processing applications, the family of networks related to the shuffle-exchange graph are interesting implementation substrates, since the channel decoding and the OFDM functionalities, which are among the most resource-intensive functionalities used in most modern standards, exhibit communication patterns which can be closely matched to such networks.

6.2 Future work

Future work in the same direction can be done in the following areas -

- De Bruijn and related networks can be evaluated for supporting flexible *turbo* decoding as well. Turbo decoding needs an interleaving function which presents a challenge for the interconnection network since the traffic pattern is random. Various routing algorithms can be evaluated on these networks in order to identify that interconnection network-routing algorithm pair which provides the best area-performance tradeoff for the interleaving function and the entire turbo decoder in general.

- Efficient ways of sharing the interconnection network among other functions in the wireless DSP unit can be investigated, in order to identify the interconnection network that provides the best area-performance tradeoff for the entire wireless DSP unit. This investigation will potentially also provide insights into the ideal interconnection network for connecting the processing units in the wireless DSP

unit, which may be realized through a *hybrid* combination of one or more of these and/or conventional interconnection networks.

- The application of *3D* interconnects and *optical* interconnects for the realization of such networks can be investigated.

List of publications from this thesis

1. Ganesh Garga, David Guevorkian, S K Nandy and H S Jamadagni, "High throughput flexible constraint length Viterbi decoders on de Bruijn, shuffle exchange and butterfly connected architectures", *Proc. of the 9th International Conference on Embedded Computer Systems: Architectures, Modelling and Simulation (SAMOS)*, July 2009, Samos, Greece. *(to appear)*

2. Ganesh Garga, M Alle, K Varadarajan, S K Nandy and H S Jamadagni, "Realizing a flexible constraint length Viterbi decoder for software radio on a de Bruijn interconnection network", *Proc. of the 10th International Symposium on System-on-chip (SOC)*, Nov 2008, Tampere, Finland.

Bibliography

[1] Easy project. http://easy.intranet.gr.

[2] Mumor project. http://www.mumor.org.

[3] Pact xpp technologies. http://www.pactcorp.org.

[4] Pleiades project. http://bwrc.eecs.berkeley.edu.

[5] D. Akopian, J. Takala, J. Saarinen, and J. Astola. Multistage interconnection networks for parallel viterbi decoders. *Communications, IEEE Transactions on*, 51(9):1536–1545, Sept. 2003.

[6] M.A. Anders, S.K. Mathew, S.K. Hsu, R.K. Krishnamurthy, and S. Borkar. A 1.9 gb/s 358 mw 16–256 state reconfigurable viterbi accelerator in 90 nm cmos. *Solid-State Circuits, IEEE Journal of*, 43(1):214–222, Jan. 2008.

[7] H. Bitterlich and H. Meyr. Efficient scalable architectures for Viterbi decoders. *Proc. of the IEEE International Conference on Application-specific architectures, systems and processors(ASAP)*, pages 89–100, 1993.

[8] M. Biver, H. Kaeslin, and C. Tommasini. In-place updating of path metrics in viterbi decoders. *Solid-State Circuits, IEEE Journal of*, 24(4):1158–1160, Aug 1989.

[9] P.J Black and T.H-Y Meng. A hardware-efficient, parallel Viterbi Decoder. *Proc. of the IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 893–896, 1990.

[10] P.J Black and T.H-Y Meng. A 140-Mb/s, 32-state, radix-4 Viterbi decoder. *IEEE Journal of solid-state circuits*, pages 1877–1885, 1992.

[11] P.J. Black and T.H.-Y. Meng. Hybrid survivor path architectures for viterbi decoders. *Acoustics, Speech, and Signal Processing, 1993. ICASSP-93., 1993 IEEE International Conference on*, 1:433–436 vol.1, Apr 1993.

[12] P.J Black and T.H-Y Meng. A 1-Gb/s, four-state, sliding block Viterbi decoder. *IEEE Journal of Solid State Circuits*, pages 797–805, 1997.

[13] M. Bóo, F. Argüello, J. D. Bruguera, R. Doallo, and E. L. Zapata. High-performance VLSI architecture for the Viterbi algorithm. *IEEE Trans. Communications*, vol. 45(2):pages 168–176, 1997.

[14] J.M. Campos and R. Cumplido. A runtime reconfigurable architecture for viterbi decoding. pages 1–4, Sept. 2006.

[15] J.R. Cavallaro and M. Vaya. Viturbo: a reconfigurable architecture for viterbi and turbo decoding. *Acoustics, Speech, and Signal Processing, 2003. Proceedings. (ICASSP '03). 2003 IEEE International Conference on*, 2:II–497–500 vol.2, April 2003.

[16] F. Chan and D. Haccoun. Adaptive viterbi decoding of convolutional codes over memoryless channels. *Communications, IEEE Transactions on*, 45(11):1389–1400, Nov 1997.

[17] Yun-Nan Chang, H. Suzuki, and K.K. Parhi. A 2-mb/s 256-state 10-mw rate-1/3 viterbi decoder. *Solid-State Circuits, IEEE Journal of*, 35(6):826–834, Jun 2000.

[18] P.M. Chau and K.J. Stephen. Scaling and folding the viterbi algorithm trellis. *VLSI Signal Processing, V, 1992., [Workshop on]*, pages 479–489, Oct 1992.

[19] J. Conan. An f8 microprocessor-based breadboard for the simulation of communication links using rate 1/2 convolutional codes and viterbi decoding. *Communications, IEEE Transactions on*, 31(2):165–171, Feb 1983.

[20] Robert Cypher and C. Bernard Shung. Generalized trace-back techniques for survivor memory management in the viterbi algorithm. *J. VLSI Signal Process. Syst.*, 5(1):85–94, 1993.

[21] William J. Dally, John Kim, and Dennis Abts. A Cost Efficient Topology for High Radix Networks. *International Symposium on Computer Architecture(ISCA '07)*, pages 126–137, 2007.

[22] H. Dawid, S. Bitterlich, and H. Meyr. Trellis pipeline-interleaving: a novel method for efficient viterbi decoder implementation. *Circuits and Systems, 1992. ISCAS '92. Proceedings., 1992 IEEE International Symposium on*, 4:1875–1878 vol.4, May 1992.

[23] J. Dielissen, Nur Engin, S. Sawitzki, and K. van Berkel. Multistandard fec decoders for wireless devices. *Circuits and Systems II: Express Briefs, IEEE Transactions on*, 55(3):284–288, March 2008.

[24] D.A. El-Dib and M.I. Elmasry. Modified register-exchange viterbi decoder for low-power wireless communications. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 51(2):371–378, Feb. 2004.

[25] D.A. El-Dib and M.I. Elmasry. Memoryless viterbi decoder. *Circuits and Systems II: Express Briefs, IEEE Transactions on*, 52(12):826–830, Dec. 2005.

[26] Portier F., Raos L., Silva A., Baudais J.-Y., Hélard J.-F., Gameiro A., and Zazo S. Transmission techniques for downlink multi-antenna mc-cdma systems in a beyond-3g context. *Journal of communication and networks*, 7(2):157–170, 2005.

[27] Xing Fang, Dong Wang, and Shuming Chen. Spva: A novel digital signal processor architecture for software defined radio. pages 856–859, 31 2008-April 4 2008.

[28] G. Fettweis, H. Dawid, and H. Meyr. Minimized method viterbi decoding: 600 mbit/s per chip. *Global Telecommunications Conference, 1990, and Exhibition.*

'Communications: Connecting the Future', GLOBECOM '90., IEEE, pages 1712–1716 vol.3, Dec 1990.

[29] G. Fettweis and H. Meyr. Parallel viterbi algorithm implementation: breaking the acs-bottleneck. *Communications, IEEE Transactions on*, 37(8):785–790, Aug 1989.

[30] G. Fettweis and H. Meyr. A 100 mbit/s viterbi decoder chip: novel architecture and its realization. *Communications, 1990. ICC '90, Including Supercomm Technical Sessions. SUPERCOMM/ICC '90. Conference Record., IEEE International Conference on*, pages 463–467 vol.2, Apr 1990.

[31] G. Fettweis and H. Meyr. Cascaded feedforward architectures for parallel viterbi decoding. *Circuits and Systems, 1990., IEEE International Symposium on*, pages 978–981 vol.2, May 1990.

[32] G. Fettweis and H. Meyr. High-rate viterbi processor: a systolic array solution. *Selected Areas in Communications, IEEE Journal on*, 8(8):1520–1534, Oct 1990.

[33] G. Fettweis and H. Meyr. Feedforward architectures for parallel viterbi decoding. *J. VLSI Signal Process. Syst.*, 3(1-2):105–119, 1991.

[34] G. Fettweis and H. Meyr. High-speed parallel viterbi decoding: algorithm and vlsi-architecture. *Communications Magazine, IEEE*, 29(5):46–55, May 1991.

[35] G. Feygin, P.G. Gulak, and P. Chow. Generalized cascade Viterbi decoder - a locally connected multiprocessor with linear speedup. *Proc. of the IEEE International Conference on Acoustics, Speech and signal processing*, pages 1097–1100, 1991.

[36] G. Feygin, P.G. Gulak, and P. Chow. A multiprocessor architecture for viterbi decoders with linear speedup. *Signal Processing, IEEE Transactions on*, 41(9):2907–2917, Sep 1993.

[37] G.D. Jr. Forney. The viterbi algorithm. *Proceedings of the IEEE*, pages 268–278, 1973.

[38] W.J. Gross, V.C Gaudet, and P.G. Gulak. Difference metric soft-output detection: architecture and implementation. *IEEE Transactions on Circuits and Systems 2: Analog and Digital Signal Processing*, pages 904–911, 2001.

[39] Glenn P. Gulak and Thomas Kailath. Locally connected VLSI architectures for the Viterbi Algorithm. *IEEE journal on selected areas in communications*, vol.6(5):527–537, 1988.

[40] P. Gulak and E. Shwedyk. Vlsi structures for viterbi receivers: Part i–general theory and applications. *Selected Areas in Communications, IEEE Journal on*, 4(1):142–154, Jan 1986.

[41] P. Gulak and E. Shwedyk. Vlsi structures for viterbi receivers: Part ii–encoded msk modulation. *Selected Areas in Communications, IEEE Journal on*, 4(1):155–159, Jan 1986.

[42] J. Hagenauer and P. Hoeher. A viterbi algorithm with soft-decision outputs and its applications. *Global Telecommunications Conference, 1989, and Exhibition. Communications Technology for the 1990s and Beyond. GLOBECOM '89., IEEE*, pages 1680–1686 vol.3, Nov 1989.

[43] Andries P. Hekstra. An alternative to metric rescaling in Viterbi decoders. *IEEE Transactions on Communications*, 37:1220–1222, 1989.

[44] David Arditti Ilitzky, Jeffrey D. Hoffman, Anthony Chun, and Brando Perez Esparza. Architecture of the scalable communications core's network on chip. *IEEE Micro*, 27(5):62–74, 2007.

[45] Lihong Jia, Yonghong Gao, J. Isoaho, and H. Tenhunen. Design of a super-pipelined viterbi decoder. *Circuits and Systems, 1999. ISCAS '99. Proceedings of the 1999 IEEE International Symposium on*, 1:133–136 vol.1, Jul 1999.

[46] P.H. Kelly and P.M. Chau. A flexible constraint length, foldable viterbi decoder.

Global Telecommunications Conference, 1993, including a Communications Theory Mini-Conference. Technical Program Conference Record, IEEE in Houston. GLOBECOM '93., IEEE, pages 631–635 vol.1, Nov-2 Dec 1993.

[47] S.-Y. Kim, H. Kim, and I.-C. Park. Path metric memory management for minimising interconnections in viterbi decoders. *Electronics Letters*, 37(14):925–926, Jul 2001.

[48] Gummidipudi Krishnaiah, Nur Engin, and Sergei Sawitzki. Scalable reconfigurable channel decoder architecture for future wireless handsets. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 1563–1568, San Jose, CA, USA, 2007. EDA Consortium.

[49] D. Lattard, E. Beigne, F. Clermidy, Y. Durand, R. Lemaire, P. Vivet, and F. Berens. A reconfigurable baseband platform based on an asynchronous network-on-chip. *Solid-State Circuits, IEEE Journal of*, 43(1):223–235, Jan. 2008.

[50] F. Thomas Leighton. Introduction to parallel algorithms and architectures. *Morgan Kaufmann publishers*, 1992.

[51] H.-D. Lin and D.G. Messerschmitt. Algorithms and architectures for concurrent viterbi decoding. *Communications, 1989. ICC '89, BOSTONICC/89. Conference record. 'World Prosperity Through Communications', IEEE International Conference on*, pages 836–840 vol.2, Jun 1989.

[52] H.-D. Lin and D.G. Messerschmitt. Architectural techniques for eliminating critical feedback paths. *Selected Areas in Communications, IEEE Journal on*, 9(5):718–725, Jun 1991.

[53] Ming-Bo Lin. New path history management circuits for viterbi decoders. *Communications, IEEE Transactions on*, 48(10):1605–1608, Oct 2000.

[54] Yuan Lin, Hyunseok Lee, Mark Woh, Yoav Harel, Scott Mahlke, Trevor Mudge, Chaitali Chakrabarti, and Krisztian Flautner. Soda: A low-power architecture for software radio. *SIGARCH Comput. Archit. News*, 34(2):89–101, 2006.

[55] Haisheng Liu, Jean-Philippe Diguet, Christophe Jego, Michel Jézéquel, and Emmanuel Boutillon. Energy Efficient Turbo Decoder With Reduced State Metric Quantization. *Proc. of the IEEE Workshop on Signal Processing Systems*, pages 237–242, 2007.

[56] R.J. McEliece and I.M. Onyszchuk. Truncation effects in viterbi decoding. *Military Communications Conference, 1989. MILCOM '89. Conference Record. Bridging the Gap. Interoperability, Survivability, Security., 1989 IEEE*, pages 541–545 vol.2, Oct 1989.

[57] B.K. Min and N. Demassieux. A versatile architecture for vlsi implementation of the viterbi algorithm. *Acoustics, Speech, and Signal Processing, 1991. ICASSP-91., 1991 International Conference on*, pages 1101–1104 vol.2, Apr 1991.

[58] H. Moussa, A. Baghdadi, and M. Jezequel. Binary de bruijn interconnection network for a flexible ldpc/turbo decoder. pages 97–100, May 2008.

[59] H. Moussa, A. Baghdadi, and M. Jezequel. Binary de bruijn on-chip network for a flexible multiprocessor ldpc decoder. pages 429–434, June 2008.

[60] H. Moussa, Amer Baghdadi, and Michel Jézéquel. On-chip communication network for flexible multiprocessor turbo decoding. pages 1–6, April 2008.

[61] Hazem Moussa, Olivier Muller, Amer Baghdadi, and Michel Jézéquel. Butterfly and benes-based on-chip communication networks for multiprocessor turbo decoding. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 654–659, San Jose, CA, USA, 2007. EDA Consortium.

[62] O. Muller, A. Baghdadi, and M. Jezequel. From parallelism levels to a multi-asip architecture for turbo decoding. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 17(1):92–102, Jan. 2009.

[63] Afshin Niktash, T. Parizi Hooman, and Nader Baghderzadeh. A reconfigurable

processor for forward error correction. *Architecture of Computing Systems (ARCS 2007)*, 4415/2007:1–13, May 2007.

[64] Seiichi Nishijima, Miyoshi Saito, and Iwao Sugiyama. Single-chip baseband signal processor for software-defined radio. *FUJITSU Sci. Tech. J*, pages 240–247, 2006.

[65] A.M. Obeid, A.G. Ortiz, and M. Glesner. A constraint length and throughput reconfigurable architecture for viterbi decoders. *Industrial Technology, 2004. IEEE ICIT '04. 2004 IEEE International Conference on*, 3:1293–1297 Vol. 3, Dec. 2004.

[66] J. Omura. On the viterbi decoding algorithm. *Information Theory, IEEE Transactions on*, 15(1):177–179, Jan 1969.

[67] I.M. Onyszchuk. Truncation length for viterbi decoding. *Communications, IEEE Transactions on*, 39(7):1023–1026, Jul 1991.

[68] C. Rader. Memory management in a viterbi decoder. *Communications, IEEE Transactions on*, 29(9):1399–1401, Sep 1981.

[69] G. K. Rauwerda, G. J. M. Smit, C. R. W. van Benthem, and P. M. Heysters. Reconfigurable turbo/viterbi channel decoder in the coarse-grained montium architecture. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'06), Las Vegas, Nevada, USA*, pages 110–116, USA, June 2006. CSREA Press.

[70] G.K. Rauwerda, P.M. Heysters, and G.J.M. Smit. Towards software defined radios using coarse-grained reconfigurable hardware. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(1):3–13, Jan. 2008.

[71] P. Robertson, P. Hoeher, and E. Villibrun. Optimal and sub-optimal maximum a posteriori algorithms suitable for Turbo decoding. *Telecommunications, European Transactions on*, (2):119–125, 1997.

[72] C.B Shung, H.D Lin, R. Cypher, P.H Siegel, and H.K Thapar. Area-efficient architectures for the viterbi algorithm. i. theory. *IEEE Transactions on Communications*, 41:636–644, 1993.

[73] C.B Shung, H.D Lin, R. Cypher, P.H Siegel, and H.K Thapar. Area-efficient architectures for the viterbi algorithm. ii. applications. *IEEE Transactions on Communications*, 41:802–807, 1993.

[74] C.B. Shung, P.H. Siegel, G. Ungerboeck, and H.K. Thapar. Vlsi architectures for metric normalization in the viterbi algorithm. *Communications, 1990. ICC '90, Including Supercomm Technical Sessions. SUPERCOMM/ICC '90. Conference Record., IEEE International Conference on*, pages 1723–1728 vol.4, Apr 1990.

[75] S.J. Simmons. Breadth-first trellis decoding with adaptive effort. *Communications, IEEE Transactions on*, 38(1):3–12, Jan 1990.

[76] J. Sparso, H.N. Jorgensen, E. Paaske, S. Pedersen, and T. Rubner-Petersen. An area-efficient topology for vlsi implementation of viterbi decoders and other shuffle-exchange type structures. *Solid-State Circuits, IEEE Journal of*, 26(2):90–97, Feb 1991.

[77] R. Tessier, S. Swaminathan, R. Ramaswamy, D. Goeckel, and W. Burleson. A reconfigurable, power-efficient adaptive viterbi decoder. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 13(4):484–488, April 2005.

[78] H.K. Thapar and J.M. Cioffi. A block processing method for designing high-speed viterbi detectors. *Communications, 1989. ICC '89, BOSTONICC/89. Conference record. 'World Prosperity Through Communications', IEEE International Conference on*, pages 1096–1100 vol.2, Jun 1989.

[79] Kou-Hu Tzou and J. Dunham. Sliding block decoding of convolutional codes. *Communications, IEEE Transactions on*, 29(9):1401–1403, Sep 1981.

[80] Kees van Berkel, Frank Heinle, Patrick P. E. Meuwissen, Kees Moerman, and Matthias Weiss. Vector processing as an enabler for software-defined radio in handheld devices. *EURASIP J. Appl. Signal Process.*, 2005(1):2613–2625, 2005.

[81] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, pages 260–269, April 1967.

[82] Timo Vogt and Norbert Wehn. A reconfigurable application specific instruction set processor for convolutional and turbo decoding in a sdr environment. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, pages 38–43, New York, NY, USA, 2008. ACM.

[83] Kuei Ann Wen and Jau Yien Lee. Parallel processing for viterbi algorithm. *Electronics Letters*, 24(17):1098–1099, Aug 1988.

[84] A. Worm, H. Michel, F. Gilbert, G. Kreiselmaier, M. Thul, and N. Wehn. Advanced implementation issues of turbo-decoders. In *In Proc. 2nd International Symposium on Turbo-Codes and Related Topics*, pages 351–354, 2000.

[85] Yufei Wu, Brian D. Woerner, and T. Keith Blankenship. Data Width Requirements in SISO Decoding With Modulo Normalization. *IEEE Transactions on Communications*, pages 1861–1868, 2001.