Single Chip Variable Rate Viterbi Decoder of Constraint Length K = 5

By
Peter Bonek

Dipl.-Ing. Technische Universität Vienna, Austria, 1991

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENT FOR THE DEGREE OF
MASTER OF APPLIED SCIENCE

in

THE FACULTY OF GRADUATE STUDIES
ELECTRICAL ENGINEERING

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

1993

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Electrical Engineering

The University of British Columbia
Vancouver, Canada

Date December 10, 1993

DE-6 (2/88)

Abstract

This thesis presents a fully self-testable integrated circuit (IC) variable-rate Viterbi decoder of constraint length $K = 5$. The chip is designed to decode convolutional codes ranging from rate 7/8 to 1/4, derived from the same rate 1/2 mother code. The architecture of the Viterbi decoder is bit-serial node-parallel. The incoming 8-level quantized channel bits are input in parallel and converted to a serial stream. This reduces the amount of interprocessor wiring area substantially, as there are only single wire connections between the add-compare-select (ACS) units. High decoding speed is still achieved because the ACS operation is carried out concurrently in each of the 16 states.

For the path memory, the register exchange technique was adopted. To reduce the ICs silicon area, the path memory is full-custom layout. For the trellis interconnections between consecutive memory stages, a novel state relabelling technique is proposed that reduces the interconnect area substantially. The area savings are accomplished by redrawing the trellis as sets of butterflies,

A major aspect of this IC is its very cost effective built-in self-test. The stuck-at fault coverage is 99% with an overhead area of only 5%, which should not lower the manufacturing yield significantly, and thus yield significant benefits.

A novel test algorithm was developed for the path memory. A specific but easy to generate test pattern is applied to the inputs. A major advantage of this deterministic test over pseudo-random techniques is that the test length is very short and, more importantly, independent of the number of states of the Viterbi decoder.

The rest of the circuit is tested by pseudo-random patterns combined with a multiple signature analysis scheme. After finding an appropriate initial state of the test pattern generator, it is possible to check for four identical signatures. Compared to checking only one signature at the end of the test session, checking four identical signatures has the advantage of reducing the probability of error escape, while avoiding complicated signature checking for four different references. Moreover, test time can be reduced as faulty chips can be discarded as soon as a signature does not match the reference. These advantageous features are accomplished with circuit overhead equal to checking only a single signature at the end of the test session. The only cost is a one-time logic simulation performed at design phase.

Table of Contents

List of Figures

Acknowledgments

I want to take the opportunity to thank my supervisors Dr. André Ivanov and Dr. Samir Kallel for their constant encouragement and valuable guidance I was fortunate to receive in the past year. Their helpful comments and critical questions made research interesting for me. I would also like to thank my colleague Yuejian Wu for his helpful discussions on ideas how to implement efficient built-in self-tests. Especially, I thank Yuejian for letting me implement his idea of multiple signature analysis, which improved the test quality dramatically. Special thanks go to Dave Gagne, our VLSI-tool guru. Without his help it would not have been possible to finish the design and layout of the chip in a reasonable time. Dave had quick solutions to almost any problem relating the use of our VLSI tool. Here, I would like to thank the Canadian Microelectronics Corporation for providing me both hardware and software tools for VLSI design and providing me access to Northern Telecom's fabrication process. I also want to thank the Austrian Government and the Austrian Federal Chamber of Commerce for their financial support that made this work possible. I would like to express my sincere thanks to my girlfriend Claudia Künzel, who paved the way to the successful completion of this project with her love and support. I am extremely grateful to my parents, who always encourage and support whatever I do. Without them nothing would have been possible.

1. Introduction

1.1 Background

A major concern in data transmission systems is how to deal with transmission errors due to noise on the channel. In recent years there has been a great interest in convolutional codes and their use in modern communications systems. Convolutional codes can be used solely for *forward error correction* (FEC), or can be incorporated into transmission systems using *automatic-repeat-request (ARQ) schemes* to ensure error-free transportation of data. The advent of high-rate punctured convolutional codes has incresed the interest in convolutional coding, as punctured codes can be readily decoded and still offer substantial coding gain. Variable-rate FEC systems use a family of punctured convolutional codes derived from the same low-rate mother code.

With a type II hybrid ARQ protocol, in addition to a block code that is used for error detection only, a second code, usually a convolutional code, is used for error correction [Kal93]. To improve the system throughput, variable code rates can be used [Hag88], [Kal90], [Kal93]. The chosen code rate depends on the channel condition, round trip delay of the data packets, and buffer size at the receiver. At the receiver end usually a Viterbi decoder, suitable for decoding variable rates, performs the error correction step. The data packet is then handed over to the block decoder for error detection. There are two basic ideas in deploying variable rates with ARQ schemes: *Adaptive Coding Rate (ACR)* ARQ protocols and *Adaptive Incremental Redundancy (AIR)* ARQ protocols [Kal93].

The Viterbi algorithm [Bha81, Hay88, Lin83] was introduced in 1967 by A. J. Viterbi as a decoding algorithm for convolutional codes. Later, Forney found that it was in fact a

maximum likelihood decoding algorithm for convolutional codes [For73]. In the past years many architectures have been proposed to increase decoding speed. Fettweis [Fet90] [Fet91] presented a 600 Mbit/s single chip Viterbi decoder for a four-state trellis. This high data rate is achieved by introducing massive pipelining and parallel processing in each of several Viterbi decoders, operating in parallel. A variable-rate Viterbi decoder was presented by Yasuda [Yas83]. The decoder is of constraint length $K = 7$ and is capable of decoding rates ranging from 1/2 up to 15/16. Single chip $K = 7$ Viterbi decoders of decoding speeds up to 25 Mbit/s are readily available commercially [Qua93] [Sta91]. However, these have the drawback of supporting only a very limited number of different rates.

The variable-rate Viterbi decoder described in this thesis will be part in a data link protocol for mobile data communications deploying different code rates depending on the channel conditions. The decoder is fully self-testable and is designed to decode any rate from 7/8 to 1/4. High code rates are obtained by puncturing (deleting bits) a 1/2 rate convolutional code periodically, whereas low rate codes are obtained by repeating encoder output bits. The advantage of using codes derived from the same 1/2 rate code is that only one 1/2 rate Viterbi decoder plus some additional control logic can be used to decode all these codes.

The design methodology adopted here is top-down. The Viterbi decoder is split into blocks of smaller and smaller size down to the gate or even the transistor level. The decoder is implemented in Northern Telecom's 1.2 μm double-metal CMOS technology. Unfortunately, only very limited silicon area was available. As a consequence the decoder is of constraint length $K = 5$. The maximum decoding speed is 12 Mbit/s channel rate. The supported modulation schemes are binary phase shift keying (BPSK) and quadrature phase shift keying (QPSK).

1.2 Contributions to Knowledge

Two major contributions can be found in this thesis. The first contribution to knowledge

deals with reducing the silicon area of register exchange path memories. In a naive approach, consecutive stages in the path memory are connected in an area wasting trellis manner. Much of the silicon area can be saved when consecutive trellis stages are redrawn as sets of butterflies. The butterfly approach reduces the number of vertical wire tracks between memory stages from N, where N is the number of states of the Viterbi decoder, between consecutive stages to $2 + 4 + ... + 2^i + ... + N$ for $\log N$ stages.

The second novelty in this thesis is a very cost-effective test scheme to test a register-exchange path memory. Instead of applying pseudo-random test vectors, the memory can be tested more efficiently with deterministic test patterns. The test patterns are generated easily from a counter, usually the test counter for pseudo-random test of other parts in the circuit. The test length of a test session is three times the length of the path memory but independent of the number of states of the Viterbi decoder. Since adjacent outputs have complement values during a complete test session, an exclusive OR tree as compactor yields zero for a fault-free path memory for every clock cycle. A bit-by-bit comparison with zero eliminates any error escape.

1.3 Outline of Thesis

Chapter two introduces the reader to the basics of convolutional codes and their decoding by the Viterbi algorithm. This chapter also introduces the concept of rate compatible convolutional (RCC) codes. The family of codes is derived from the highest rate code obtained from a mother rate 1/2 code by adding back previously deleted bits.

Chapter three is devoted to the decoder realization. First it discusses some possible decoder architectures and gives an overview of metric normalization techniques. Then, the two realizations of a path memory in use are introduced and compared. Finally, this chapter discusses different implementation possibilities. Programmable logic devices, field

programmable gate arrays, and application-specific integrated circuits are described and compared.

Chapter four presents solutions to the implementation of the basic blocks of the variable-rate Viterbi decoder. One section in this chapter is devoted to a new approach for wiring the memory elements in the path memory. Redrawing consecutive stages of trellises as sets of butterflies achieves considerable silicon-area savings in custom-layout path memories.

Chapter five starts with an introduction to built-in self-test. A later section describes the implementation of a minimal hardware overhead multiple signature scheme that reduces error escape and possibly test time. This chapter also presents a new, very cost effective test algorithm for a register exchange path memory by deploying a deterministic test instead of a pseudo-random test.

Chapter six gives some practical details about the fabricated chip and instructions on how to operate the Viterbi decoder. Finally, chapter seven summarizes this thesis.

2. Convolutional Encoding and Viterbi Decoding

2.1 Convolutional Codes

A convolutional code is a type of code where the encoder has memory. The n encoder outputs not only depend on the k inputs at that time unit but also on m previous inputs. Hence, the encoder has a *memory order* of m. If $k > 1$, m is defined as the maximum of all k feedforward shift registers. The code is called an *(n, k, m)* convolutional code [Lin83]. The ratio $R = k/n$ is called the *code rate*. Without loss of generality, the following discussion is restricted to $1/n$ rate codes. As an example, consider the encoder for a *(2, 1, 3)* binary code, which is shown in Fig. 2.1. The binary *input sequence* $\mathbf{x} = (x_0\ x_1\ x_2\ ...)$ enters the encoder

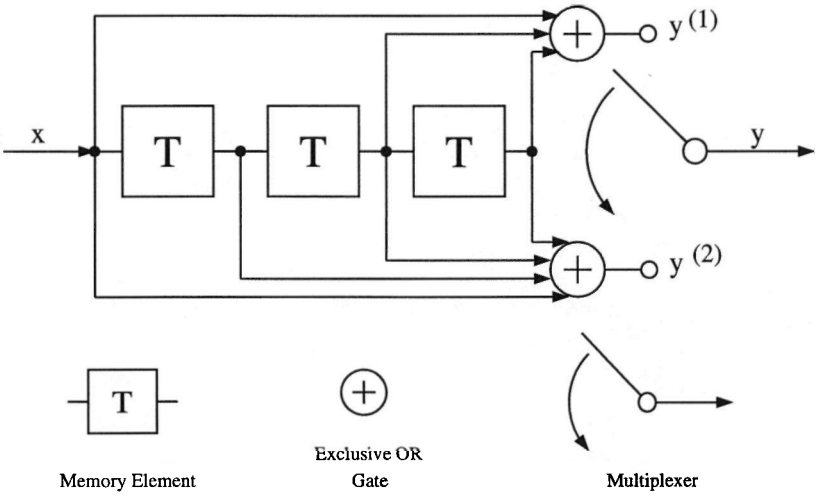


Figure 2.1 Encoder for a *(2, 1, 3)* a binary code.

one bit at a time. The two *output sequences* $\mathbf{y}^{(1)} = (y_0^{(1)}\ y_1^{(1)}\ y_2^{(1)}\ ...)$ and $\mathbf{y}^{(2)} = (y_0^{(2)}\ y_1^{(2)}\ y_2^{(2)}\ ...)$ are obtained by discrete convolution of the input sequence with the two encoder impulse responses, whereby the impulse responses are the observed output sequences of the

input sequence **x** = (1 0 0 ...). In general, there are n output sequences. As the encoder is of memory m, the input can influence the output for at most $m + 1$ time units, called the *constraint length K*. The impulse responses, called *generator sequences*, are written $\mathbf{g}^{(1)} = (g_0^{(1)}\ g_1^{(1)} \ldots\ g_m^{(1)})$, $\mathbf{g}^{(2)} = (g_0^{(2)}\ g_1^{(2)} \ldots\ g_m^{(2)})$, ..., $\mathbf{g}^{(n)} = (g_0^{(n)}\ g_1^{(n)} \ldots\ g_m^{(n)})$. For the encoder in Fig. 2.1,

$$\begin{aligned} \mathbf{g}^{(1)} &= (1\ 0\ 1\ 1) \\ \mathbf{g}^{(2)} &= (1\ 1\ 1\ 1). \end{aligned} \tag{2.1}$$

The encoding equations can now be written as

$$\begin{aligned} \mathbf{y}^{(1)} &= \mathbf{x} * \mathbf{g}^{(1)} \\ \mathbf{y}^{(2)} &= \mathbf{x} * \mathbf{g}^{(2)} \\ &\cdots \\ \mathbf{y}^{(n)} &= \mathbf{x} * \mathbf{g}^{(n)}, \end{aligned} \tag{2.2}$$

where * denotes discrete convolution and all operations are in modulo 2 arithmetic. For all $l \geq 0$ the discrete convolution is given by

$$y_l^{(j)} = \sum_{i=0}^{m} x_{l-i} g_i^{(j)} = x_l g_0^{(j)} + x_{l-1} g_1^{(j)} + \ldots + x_{l-m} g_m^{(j)}, \quad j = 1, 2, \tag{2.3}$$

where $x_{l-i} \triangleq 0$ for all $l < i$. The transmitted output sequence, the *code word*, **y** is obtained by multiplexing the n output sequences $\mathbf{y}^{(1)}$, $\mathbf{y}^{(2)}$, ..., $\mathbf{y}^{(n)}$ into a single sequence. One can also rewrite the encoding equations as a matrix multiplication

$$\mathbf{y} = \mathbf{x}\mathbf{G}, \tag{2.4}$$

where **G** is called the *generator matrix* and formed by interlacing the n generator sequences $\mathbf{g}^{(1)}$, $\mathbf{g}^{(2)}$, ..., $\mathbf{g}^{(n)}$ and arranging them in the following matrix

$$\mathbf{G} = \begin{bmatrix} g_0^{(1)} g_0^{(2)} \ldots g_0^{(n)} & g_1^{(1)} g_1^{(2)} \ldots g_1^{(n)} & g_2^{(1)} g_2^{(2)} \ldots g_2^{(n)} & \cdots & g_m^{(1)} g_m^{(2)} \ldots g_m^{(n)} & \\ & g_0^{(1)} g_0^{(2)} \ldots g_0^{(n)} & g_1^{(1)} g_1^{(2)} \ldots g_1^{(n)} & \cdots & g_{m-1}^{(1)} g_{m-1}^{(2)} \ldots g_{m-1}^{(n)} & g_m^{(1)} g_m^{(2)} \\ & & \ddots & & & \ddots \end{bmatrix}. \tag{2.5}$$

The blank elements are all zero and all operations are in modulo 2 arithmetic.

To find the output sequence, we can either compute the discrete convolution, calculate the matrix multiplication, or simply shift the input sequence through the encoder. Let

$$\mathbf{x} = (1\ 0\ 0\ 1\ 1). \tag{2.6}$$

Encoding **x** = (1 0 0 1 1) by the encoder from Fig. 2.1, yields

$$\mathbf{y} = (11\ 01\ 11\ 00\ 10\ 01\ 11\ 11). \tag{2.7}$$

For reasons of clarity, since one input bit produces two output bits, the output sequence is represented as a stream of bit pairs. For this finite length L information sequence, the corresponding output sequence is of length $n(L+m)$, where the last nm outputs are generated after the last information bit has entered the encoder and is followed by zeros until the encoder is cleared. This *true code rate* is given by $L/n(L+m)$. In practical applications, where the information sequences are long compared to the memory order, i. e., $L \gg m$, the reduction in code rate due to clearing the encoder is negligible.

2.2 Convolutional Encoder

Since the encoder is a sequential circuit of memory m, we can describe its operation by a state diagram. The state of the encoder is defined as the contents of the shift register. For binary logic, the total number of states is $N = 2^m$. Figure 2.2 shows the state diagram of the encoder in Fig. 2.1. The states are labelled S_0, S_1,...,$S_{2^M-1}$with the arrows indicating the

possible state transitions. The state transitions are labelled by their input/output pairs. The

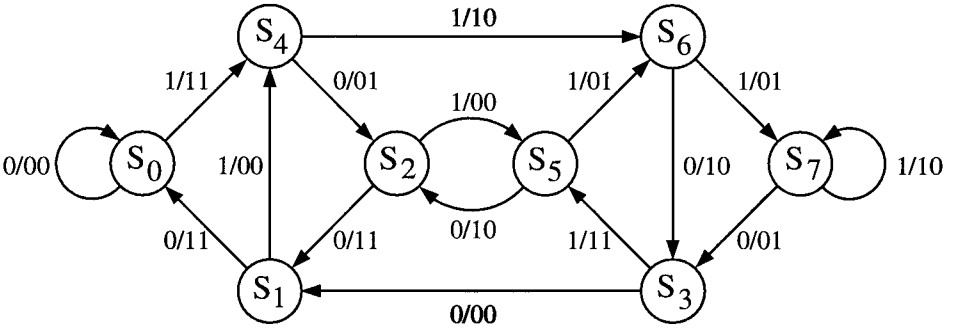


Figure 2.2 State diagram of the (2,1,3) encoder of Fig 2.1.

state diagram in Fig. 2.2 is typical for any $1/n$ encoder with binary inputs. Each state is reached from two states and is the origin of two state transitions to successor states since one bit is shifted in and one bit is shifted out of the encoder memory at every clock cycle. However, a more convenient way of displaying the state diagram, which becomes essential in the decoding process, is to expand the state diagram in time. The resulting structure that represents each discrete time step k with a state diagram is called a *trellis diagram* (Fig. 2.3). The branches of a trellis are labelled with the encoder's output corresponding to each state

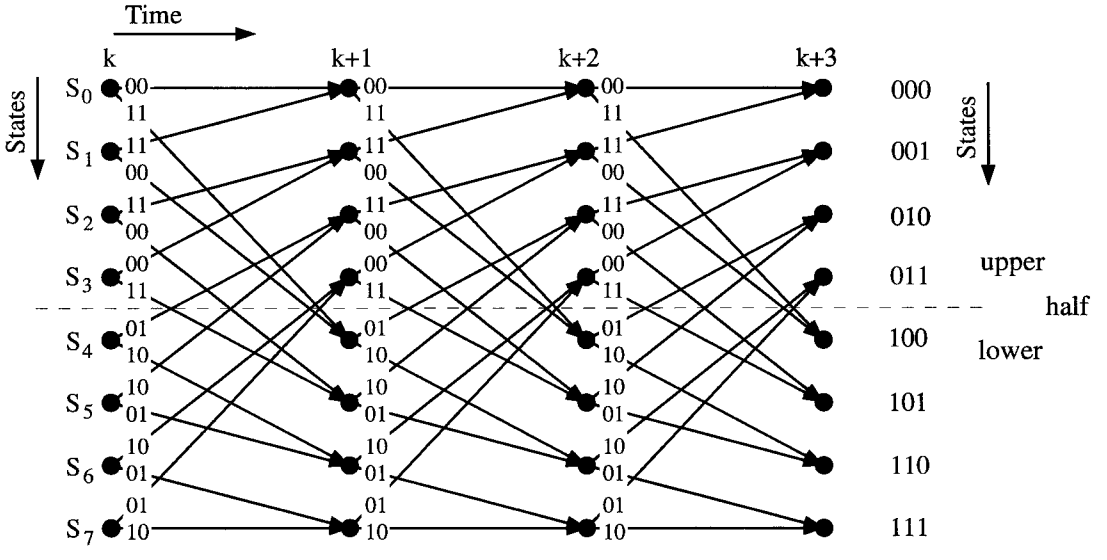


Figure 2.3 Trellis diagram for the encoder of Fig. 2.1.

transition. What becomes apparent is that all "upward" branches in the trellis end in the upper half where the binary representation of the states start with a "0" (000, 001, ..., 011). This indicates that these upper states have been reached by encoding a "0" most recently. Conversely, the lower half of the states, starting with "1", is reached only by "downward" branches. A "1" has been encoded. With the help of a trellis diagram encoding "by hand" becomes very easy. For example, the message **x** = (1 1 0 1 0 ...) translates into moving "down, down, up, down, up, ..." in the trellis diagram of Fig. 2.4, assuming the encoder was initially cleared. The output sequence is obtained by just reading out the corresponding

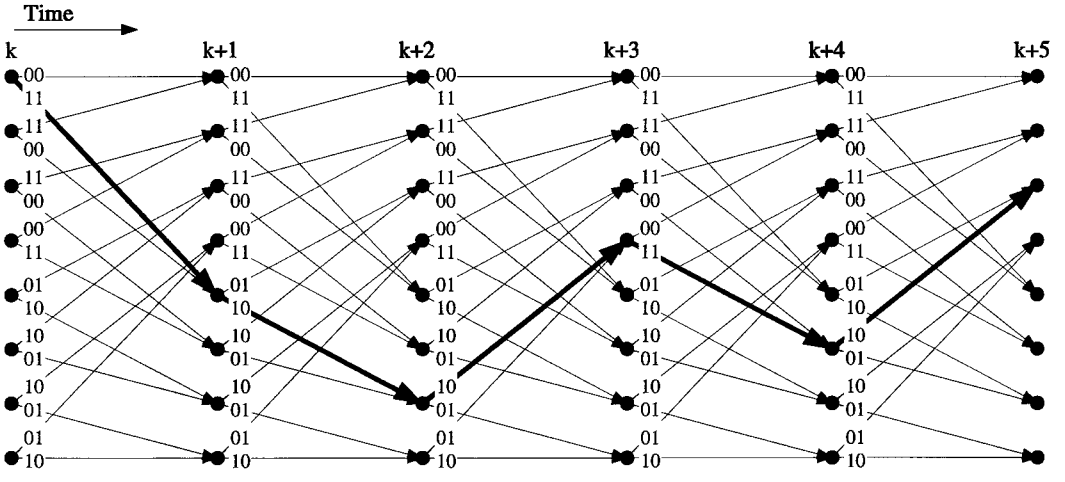


Figure 2.4 Encoding of the input sequence **x** = (1 1 0 1 0 ...).

branch labels yielding **y** = (11 10 10 11 10 ...).

The above discussion of convolutional codes and encoders considered only binary symbols. In the case of ν-ary symbols, the same theory applies. The encoder has ν^m states. In any $1/n$ code, ν branches enter and leave each state in the state diagram or the trellis diagram [Rad81]. This thesis will consider binary input sequences only. It is left to the reader to generalize for ν-ary symbols if necessary.

2.3 Code Evaluation

Why send more bits over the channel than are actually needed to represent a message? Unfortunately, noise on the channel may change a "1" into a "0" or vice versa. If all bit combinations were valid code words, noise could change the original message into another one. Thus, by introducing redundancy (in the example of Fig. 2.1 one input bit becomes two output bits) and not allowing all possible bit combinations as valid output sequences (starting from S_2, 11 11 11 11 is not an allowed sequence), enables the decoder to correct bit errors randomly introduced on the channel to a certain degree. The figure of merit defined in convolutional codes is the *free distance*

$$d_{free} \triangleq \min \{ d(y', y'') : x' \neq x'' \}, \tag{2.8}$$

where y' and y'' are the code words corresponding to the information sequences x' and x'', respectively [Lin83]. The distance $d(\bullet,\bullet)$ of two code words is defined as the number of bits where they differ. Since a convolutional code is a linear code, d_{free} is also the minimum-weight output sequence produced by any nonzero input sequence. In the trellis diagram d_{free} is the minimum weight of all paths that diverge and converge with the all-zero state S_0. The weight is defined as the number of non-zero components of y. In our example code, the sequence **x** = (1 1 0 0 0 ...) produces the minimum-weight path with a weight of six (Fig. 2.5). In general d_{free} increases with increasing encoder memory m [Hac89]. The bigger d_{free} is, the more channel bit errors on the channel are necessary to change a transmitted sequence into another valid sequence that hence yields undetectable errors.

2.4 Punctured and Repetition Codes

2.4.1 Punctured Convolutional Codes

Usually, the design of an error correction coding system requires the selection of a fixed rate code with a certain error correction capability depending on data protection requirements

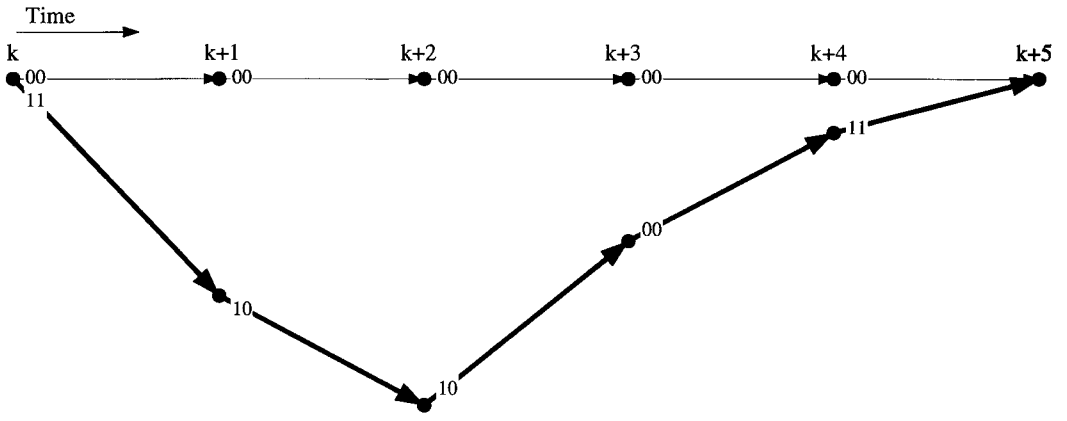


Figure 2.5 The input sequence x = (1 1 0 0 0 ...) produces the minimum-weight path with a weight of six.

and the channel noise. Ideally, one may wish to change the code rate depending on the current channel noise and/or the information significance [Hag88, Kal90]. The idea is to use a mother rate $1/n$ convolutional code and periodically puncture the code with period P. Puncturing a code means deleting certain bits of the $1/n$ rate code by following a specific perforation pattern. This yields a family of rate-compatible punctured convolutional (RCPC) codes with decreasing rates $P/(P+l)$, where l can vary from 1 to $(n-1)P$. Rate compatibility requires that all code bits of a high rate code are used in the lower rate codes. Figure 2.6a shows the trellis structure of a standard rate R=2/3, m=2 code and Fig. 2.6b shows its equivalent punctured 1/2 rate code. The X's indicate that those bits are deleted in the encoder and not sent through the channel.

The construction of a family of RCPC codes from a known "best" high rate $(n-1)/n$ punctured convolutional code, obtained from a 1/2 rate code, is straightforward [Kal90]. "Best" in this case means having the best error performance, not necessarily the largest free distance d_{free} [Cai79]. As the error performance highly depends on the deleting bit positions, a "best" high rate code is found by examining all possible bit deletion positions. Tables of high rate $(n-1)/n$ punctured codes with maximum free distance for $3 \leq n \leq 14$ and memory

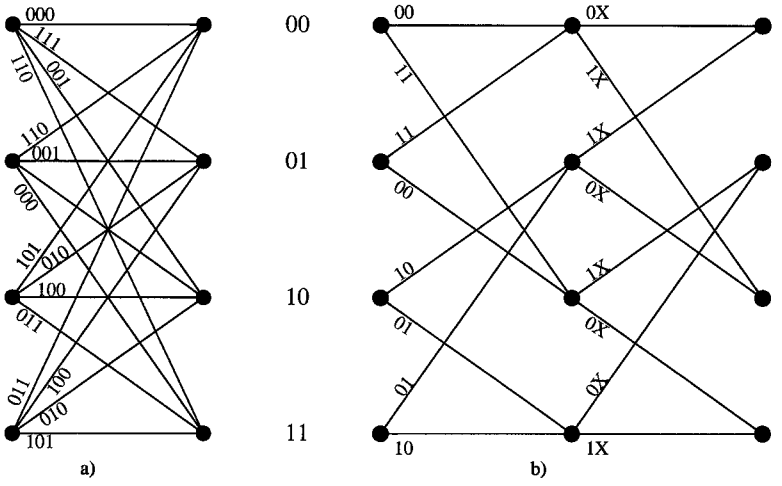


Figure 2.6 a) Trellis of an R=2/3, m=2 code b) its equivalent punctured R=1/2 code.

$2 \leq m \leq 8$ and $3 \leq n \leq 17$ and memory $m = 6$ are given in [Yas84] and [Yas83], respectively. Starting with the $(n\text{-}1)/n$ high rate code, rate-compatible lower rate codes are obtained by adding back the bits that were initially deleted to get the $(n\text{-}1)/n$ rate code. The representation of the perforation pattern is usually in the form of a matrix **P**, the *perforation matrix.* The following is an example of a perforation matrix of a rate 4/5 code:

$$\mathbf{P_1} = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}. \tag{2.9}$$

The matrix is of size $n \times P$, where a zero means that this code bit is not transmitted. The two (n in general) represent the two (n) branches in the encoder that are multiplexed to form a single output sequence (recall Fig. 2.1). In the above example matrix both code bits of the first information bit are transmitted because the first column contains two "1"s. For the second and the third information bits only the first code bit, which is the code bit of the upper branch of the encoder, is transmitted. Finally, for the fourth information bit, only the second code bit, from the lower branch, is output to the channel. This procedure repeats until the end of the message is reached. Suppose the error correction capability is not sufficient

to overcome the current channel noise. Filling up the "0"s in $\mathbf{P_1}$ with "1"s,

$$\mathbf{P_2} = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix}, \quad \mathbf{P_3} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{bmatrix}, \quad \mathbf{P_4} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}, \tag{2.10}$$

finally results in the mother 1/2 rate code. This procedure generates a family of rate compatible codes with incremental redundancy. Determining their performance must be achieved through simulations. An effective system could make efficient use of the channel by only retransmitting the newly added bits and combining them with the previously received erroneous message [Kal90].

2.4.2 Repetition Convolutional Codes

Equivalently to constructing a family of RCPC codes by adding ones into the perforation matrix of the (n-1)/n high rate code, a family of rate-compatible repetition convolutional (RCRC) codes can be generated by replacing "1"s by higher values ("2", "3",...) without any limit [Kal90]. The matrix that represents an RCRC code is called a *repetition matrix* and is denoted $\mathbf{Q}$. As an example, a 4/11 rate code would be represented by

$$\mathbf{Q_1} = \begin{bmatrix} 2 & 1 & 2 & 1 \\ 2 & 1 & 1 & 1 \end{bmatrix}, \tag{2.11}$$

where a "2" means that this particular code bit is sent twice. In $\mathbf{Q_1}$, the first and every other fourth information bit will produce four channel bits because the first column in the repetition matrix is 2-2 (repeat both the upper and the lower branch of the encoder). Two repetition codes obtained from the same original code are said to be rate-compatible if every element of the repetition matrix of the lower rate code is equal or greater than its corresponding element in the repetition matrix of the higher rate code. A 4/13 rate code, rate-compatible to the code of $\mathbf{Q_1}$, could be of the form:

$$\mathbf{Q_2} = \begin{bmatrix} 2 & 1 & 2 & 1 \\ 3 & 2 & 1 & 1 \end{bmatrix}. \tag{2.12}$$

2.5 Viterbi Decoding

The Viterbi algorithm [Bha81, Hay88, Lin83] was introduced in 1967 by A. J. Viterbi as a decoding algorithm for convolutional codes. Later, Forney found that it was in fact a maximum likelihood decoding algorithm for convolutional codes [For73].

Let $\mathbf{x} = (x_0, .., x_{L-1})$ be the information sequence of L bits encoded into the code word $\mathbf{y} = (\mathbf{y}_0, .., \mathbf{y}_{L+m-1})$, where $\mathbf{y}_i = y_{1i}, y_{2i}, ..., y_{ni}$, of length $n(L+m)$. The received sequence is denoted by $\mathbf{r} = (\mathbf{r}_0, .., \mathbf{r}_{L+m-1})$ with $\mathbf{r}_i = r_{1i}, r_{2i}, ..., r_{ni}$. As a maximum likelihood decoder, the Viterbi algorithm attempts to find the sequence $\hat{\mathbf{y}}$ that is closest to the received sequence $\mathbf{r}$. Assuming equiprobable input data sequences, the decoder chooses the path through a trellis that maximizes P(**r**/**y**), where the conditional probability P(**r**/**y**) is the likelihood of the received sequence **r**, given that **y** was sent. In a discrete memoryless channel (DMC) [Lin83], every received symbol $\mathbf{r}_i$ is only dependent on the corresponding symbol $\mathbf{y}_i$, and hence,

$$P(\mathbf{r}/\mathbf{y}) = \prod_{i=0}^{L+m-1} P(\mathbf{r}_i/\mathbf{y}_i) = \prod_{j=0}^{n(L+m-1)} P(r_j/y_j). \tag{2.13}$$

Generally, it is more convenient to use the logarithm of the likelihood function (3.1) because the product turns into a sum, which can be more easily implemented. Since the logarithm is a monotonically increasing function, it does not alter the final result and (3.1) becomes the log-likelihood function

$$\log P(\mathbf{r}/\mathbf{y}) = \sum_{i=0}^{L+m-1} \log P(\mathbf{r}_i/\mathbf{y}_i) = \sum_{j=0}^{n(L+m-1)} \log P(r_j/y_j). \tag{2.14}$$

The negative log-likelihood function –log P(**r**/**y**) is called the *path metric* associated with the path **y**. The terms –log P($\mathbf{r}_i$/$\mathbf{y}_i$) are called the *branch metrics* [Mag90]. Let the terms –log $P(r_j/y_j)$ be called the *partial branch metrics*, i. e., the metrics from each of the channel symbols. The problem of maximizing the likelihood function has been transformed into minimizing the path metric.

For a binary symmetric channel (BSC) or hard-quantized channel, the received sequence $\mathbf{r}$ is binary as shown in Fig. 2.7a, where p denotes the channel transition probability. Minimizing the path metric is equivalent to minimizing the Hamming distance, the total number of bits of $\hat{\mathbf{y}}$ that differ from those of $\mathbf{r}$. In the case of a soft-quantized DMC, which can be seen in Fig. 2.7b (P(0), P(1),.., P(7) are transition probabilities), the log-likelihood function must be used. However, in terms of implementation, since metrics have to be

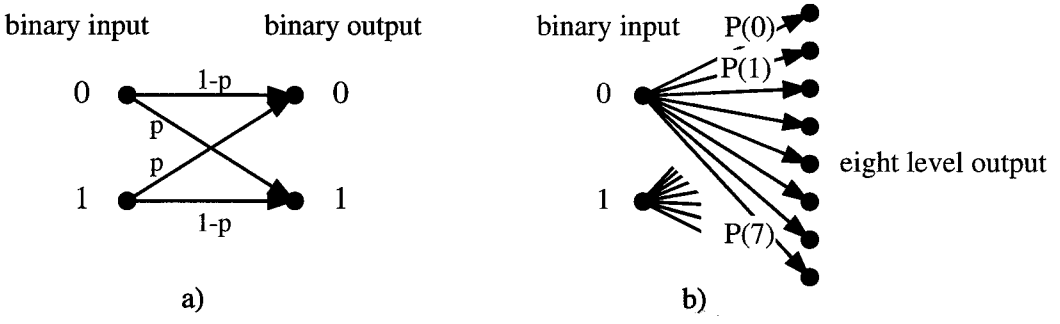


Figure 2.7 a) Hard-quantized channel (BSC) b) Soft-quantized channel.

represented in binary form, it is more convenient to use integers as metrics. In order to round off, the log $P(\mathbf{r_i}/\mathbf{y_i})$ metric can be replaced by α_1 log $P[(\mathbf{r_i}/\mathbf{y_i}) + \alpha_2]$, where α_1 is a real number and α_2 a non-negative real number [Bha81]. Yasuda [Yas81] showed that the decoder performance is rather insensitive to the particular choice of the α's and the threshold spacing of the quantization. Simulations [Yas81] [Mag90] showed that an eight-level quantized channel outperforms a BSC by about 2dB. Thus, achieving the same bit error rate (BER) requires about 2dB less signal-to-noise ratio SNR. Increasing the number of quantization levels to infinity, an ideal soft-decision DMC, gains only another 0.25dB. A typical metric table is shown in Tab. 2.1.

By definition, maximum likelihood decoding requires the comparison of the received sequence with all possible transmitted sequences before making a decision. For an L-bit-long binary input sequence, 2^L accumulated metrics have to be compared, and the sequence with the lowest metric is chosen as the estimate $\hat{\mathbf{y}}$. However, the exponential increase of

		Channel Output							
Partial Branch Metrics γ		0	1	2	3	4	5	6	7
Channel Input	0	0	1	2	3	4	5	6	7
(encoder output)	1	7	6	5	4	3	2	1	0

Table 2.1 Metric table for a binary input-eight-level output DMC.

decoding effort with L makes a "brute force" maximum likelihood decoder impractical to implement. One of the more practical but suboptimum algorithms is sequential decoding [Lin83]. In sequential decoding, the decoding of the received message is performed on one branch at a time. Starting from the root node of the tree, the algorithm selects that branch that has the lowest accumulated metric [Hac89].

With Viterbi decoding, there is a fixed number of states to be examined, one for each state of the encoder. The number of states is independent of the sequence length L, but grows exponentially with the constraint length K. This limits practical codes to short constraint lengths ($K \leq 8$), although a Viterbi decoder of $K = 15$ has been reported recently [Col92]. For each state of the trellis, the path metric is updated by adding the branch metrics of the entering branches, whereby the branch metrics are obtained by comparing the received symbol with the expected symbols for all possible state transitions (Fig. 2.8). For better readability the following examples will be restricted to constraint length $K = 3$. The decoder stores the lower path metric and keeps its associated path in the path memory as the "survivor" and discards the other one. This is done in each state and for every time unit. Hence, the decoder keeps only $N = 2^{K-1}$ paths and path metrics over the entire message length. In the case of a tie in a state, i. e., where both updated path metrics have the same value (state 10 in Fig. 2.8), there are two maximum likelihood paths through this node. The survivor is usually selected arbitrarily. At the end of the message, a tail is appended to clear the encoder and bring the Viterbi decoder into state S_0. As the trellis is only extended into

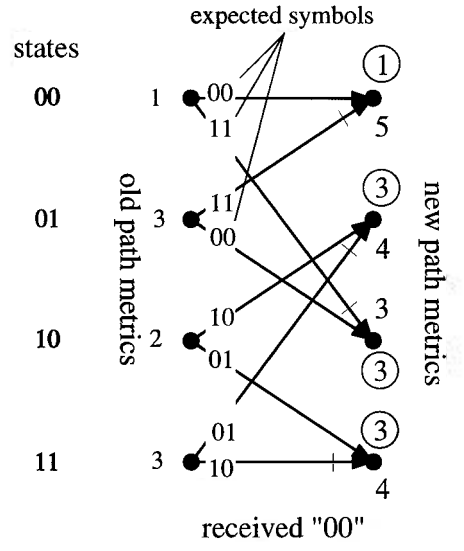


Figure 2.8 Updating the path metrics by adding the branch metrics.

branches corresponding to a "0", there is finally a single survivor, the decoded maximum likelihood sequence. A decoding example can be seen in Fig. 2.9. The "|" on the branches

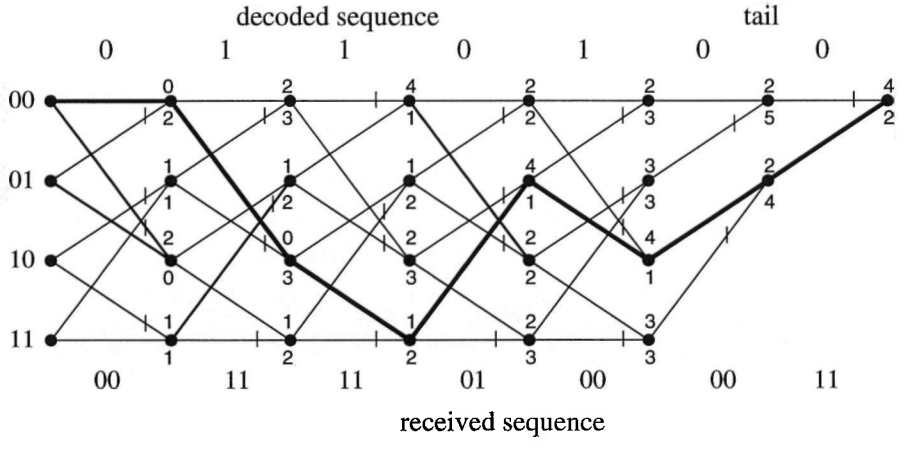


Figure 2.9 Maximum likelihood Viterbi decoding example. A tail is appended to clear the encoder.

indicates that those paths are discarded.

The great advantage of the Viterbi decoder is the constant number of operations at successive levels. These are always of the same nature and are not very sophisticated. The main disadvantage however is that it requires to wait for the tail to get a single survivor.

For long information sequences, this implies a long delay before delivering the first symbol. On the other hand, using only short messages greatly reduces the code rate unnecessarily. Simulations have shown that all N surviving paths stem from a single node four to five constraint lengths earlier with high probability [Lin83]. In the example in Fig. 2.10, the survivors (solid bold and dashed lines) merge even in less than four times the constraint length. This also solves the problem of a huge path memory in the case of very long

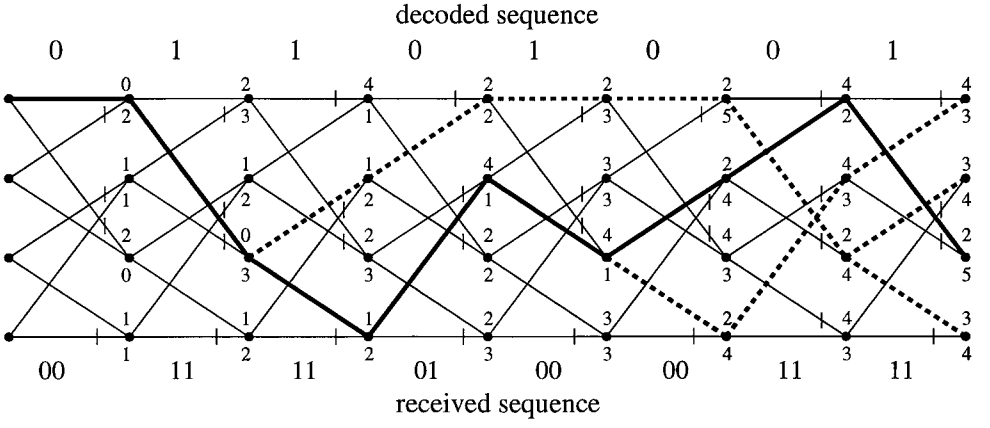


Figure 2.10 All survivors stem from the same sequence.

messages. There is no need to store 2^{K-1} L-bit paths plus their metrics but only paths where $\tau \ll L$ and τ is about four to five times the constraint length. Hence, after τ decoding steps the path memory is full and a decision has to be made. Among several decision-making strategies, there are [Sny83]:

- The oldest (first) bit from an arbitrary survivor is selected;
- A majority vote is taken from all $N = 2^{K-1}$ first information bits, that is the oldest bit that appears most often in all paths is output;
- The survivor with the lowest metric is chosen, and its oldest bit is output (solid bold line in Fig. 2.10).

Clearly, the above is not a maximum likelihood decoder anymore. However, the performance degradation is negligible if τ is not too short. In fact, a Viterbi decoder may start decoding from any state, and after producing about four to five times the constraint length of unreliable information bits, will resynchronize itself.

Viterbi decoding of punctured high rate codes is hardly more complex than decoding the mother rate 1/2 code. In fact, using punctured codes greatly reduces the complexity of decoding high rate codes [Cai79]. For the rate 2/3 code in Fig. 2.6a we notice that four branches (2^k in general) enter each node of the trellis. Thus, at every time unit and for each node four branch metrics have to be added to the old path metrics, and the minimum out of four updated path metrics has to be found and retained. For higher code rates this becomes definitely impractical. However, in Fig. 2.6b, which represents the same rate 2/3 code as Fig. 2.6a, again only two branches enter each state in the trellis diagram as in any $1/n$ rate code. Given that the decoder knows the perforation matrix, the decoding is performed on the trellis of the original mother code. The only necessary modification consists of assigning the same branch metric (usually "0") to all state transitions or simply skip any metric increment whenever a code bit was not sent.

An unwanted side effect in decoding high rate codes is an enormous increase in path memory length with increasing code rate. The survivors do not merge after processing 4 to 5 constraint lengths of bits as in the 1/2 rate code any more, but at up to 34 times the constraint length for a 16/17 rate code [Yas83]. Moreover, with increasing code rate (= decreasing redundancy) the free distance decreases and the coding gains will become smaller. However, Yasuda found that at a BER = 10^{-6}, a coding gain of 3dB versus an uncoded system is still achievable with a rate 15/16, $K = 7$ code .

The decoding procedure for low-rate repetition codes again is very simple with the use of the trellis of the mother 1/2 rate code. For the branch metric calculation however, similarly

to assigning no branch metric to punctured symbols in high-rate codes, it is necessary to assign multiple branch metrics to one information bit in the case of repetition codes. Instead of the expected symbols 00, 01, 10, 11 (see Fig. 2.8), for example for the third column in $\mathbf{Q_1}$ and $\mathbf{Q_2}$, which is 2-1, 000, 001, 110, 111 are expected before selecting the survivors. In other words, instead of the usual two channel bits in the 1/2 rate code, three have to be taken into account.

Summarizing, it is easy to generate high-rate punctured and low-rate repetition convolutional codes from a rate 1/2 code. The encoder only has to be succeeded by a pattern, either perforation or repetition, represented in the *code matrix* **C**.

3. Viterbi Decoder Realization

The Viterbi algorithm can be split into three basic blocks (Fig. 3.1): branch metric unit

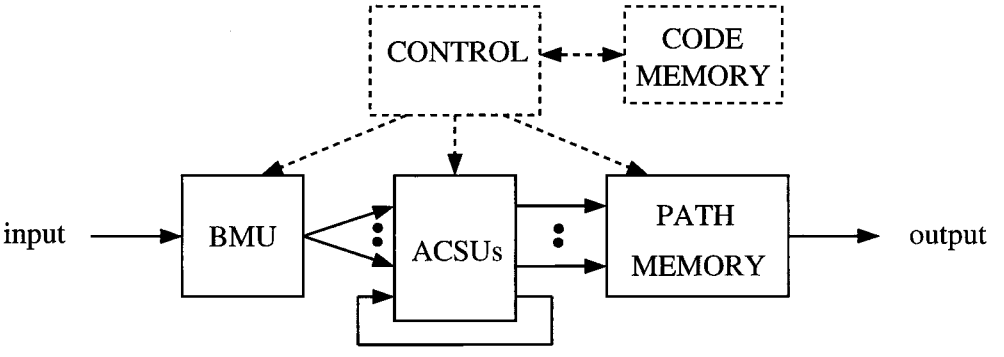


Figure 3.1 The basic blocks of a Viterbi decoder.

(BMU), add-compare-select units (ACSUs), and path memory. Depending on the particular architecture that is chosen for the decoder there will always be some control logic. A variable-rate Viterbi decoder also requires a memory to store the current code matrix **C**, i. e., a code memory. The BMU calculates the branch metrics, a measure of likelihood that the received symbol matches the transmitted symbol for each branch of the trellis. In the ACSU, the metrics associated with each state of the trellis, i. e., the path metrics, are updated by adding the new branch metrics, comparing the updated path metrics, and selecting the smaller as the surviving path metric. The path memory stores the information bits corresponding to the surviving metric.

3.1 Branch Metric Unit

The branch metric unit (BMU) has to evaluate the incoming channel symbols and compare them to the expected symbols for each of the possible state transitions. It assigns

a value of likelihood, called the *branch metric*, that the incoming and the expected symbols match. The BMU can be implemented as either a small *read-only memory* (ROM) or a correlator [Gul86]. For a BSC, where the output of the channel is either "0" or "1", the branch metric is simply the Hamming distance between the incoming and the expected symbols. The branch metrics for a finer quantized, soft-decision channel are either read from a table like Tab. 2.1 or can be computed as the squared Euclidean distance [Shu91]. In principle, there is no problem in calculating the branch metrics, however, the complexity of the implementation will highly depend on the different modulation schemes.

3.2 Decoder Architectures

3.2.1 Bit-Parallel Node-Parallel Architecture

A bit-parallel node-parallel (fully parallel) architecture, as can be seen in Fig. 3.2, will be the fastest but also the most silicon area consuming realization. The BMU calculates the branch metrics from the quantized input channel symbols, with a branch metric word width of *bmw* bits. The BMU feeds the branch metrics in parallel to the ACSUs, one for each state of the trellis. The two feedback loops at the outer ACSUs symbolize a trellis-like

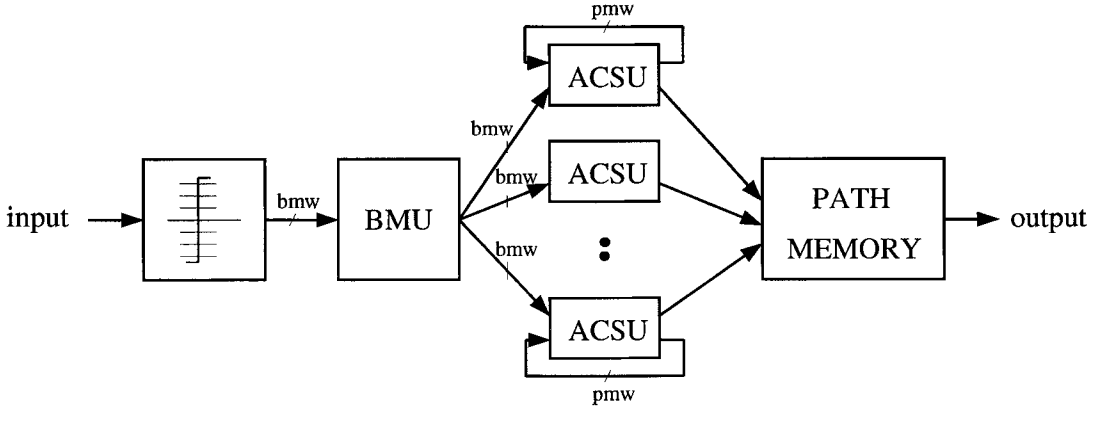


Figure 3.2 Block diagram of a bit-parallel node-parallel Viterbi decoder.

connection between the ACSUs, where the outputs of the ACSUs feed back to the inputs of

the successor states according to the trellis diagram. The path metrics have a path metric word width of *pmw* bits. The decoding strategy is very fast, as we get an output bit at every clock cycle. Note, that not only are the branch metric connections *bmw*-bit busses, but also that the trellis wired connections between the ACSUs are *pmw*-bit busses. However, in a VLSI implementation, such interconnect area could exceed the processing area and thus be a limiting factor.

3.2.2 Bit-Parallel Node-Serial Architecture

The opposite extreme to the fully parallel decoder is to share one ACSU for all nodes. The path metrics are updated sequentially. For each node in the trellis, at every cycle, two path metrics have to be fetched from a central metric memory (Fig. 3.3), the branch metrics added, and the lower resulting metric stored back in the memory. Of course, this results in a comparatively slow decoder, which moreover needs additional control logic for the scheduling. However, the area savings will be substantial with only one ACSU and no wide trellis wired busses.

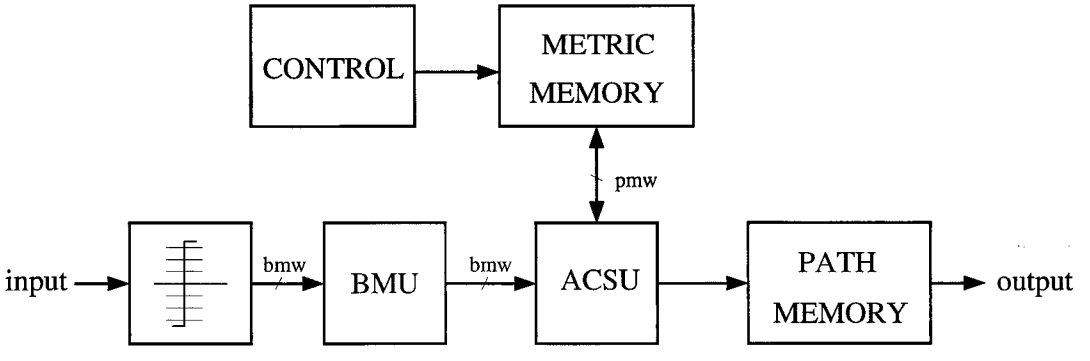


Figure 3.3 Block diagram of a bit-parallel node-serial architecture.

3.2.3 Shared Nodes Architecture

A compromise between the small silicon area of the node-serial solution and the fast decoding speed of the fully parallel implementation is to share several ACSUs. In this case, at every clock cycle, more than one but less than 2^N ACSUs perform the ACS operation.

Performing the ACS operation partly in parallel speeds up the node-serial implementation considerably, but requires less silicon area than the fully parallel architecture. Some interesting solutions are given in [Gul88]. Unfortunately, a considerable amount of control logic is needed to share the ACSUs.

3.2.4 Bit-Serial Node-Parallel Architecture

For a bit-serial node parallel architecture (Fig. 3.4), the incoming quantized channel symbols, are input in parallel and converted to a serial stream. This reduces the amount of interprocessor wiring area substantially, as there are only single wires connecting the BMU and the 2^N ACSUs. Moreover, the trellis connection between the ACSUs, symbolized by the

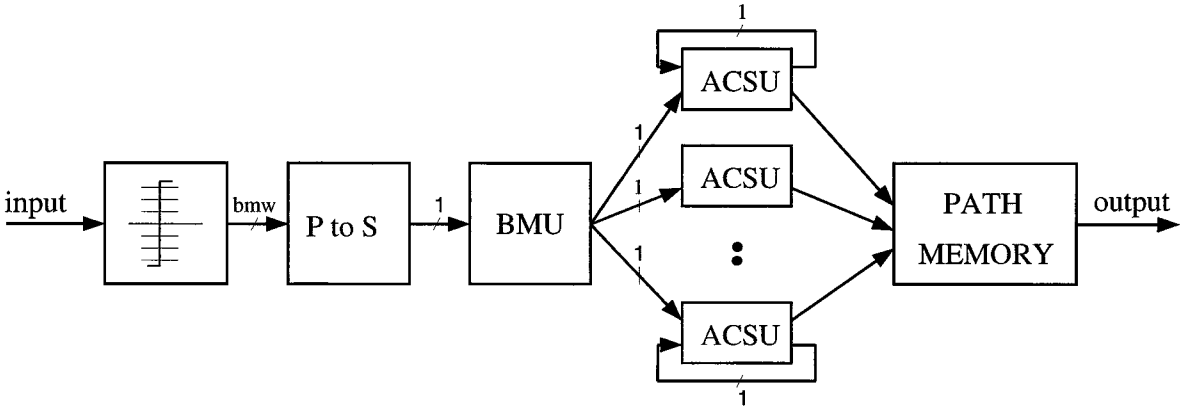


Figure 3.4 Block diagram of a bit-serial node-parallel architecture.

two feedback loops at the outer ACSUs in Fig. 3.4, are 1-bit lines. Note that using the bit-serial approach does not only save interconnect area, but also considerable computation area. Instead of using *pmw*-bit parallel adders and comparators, the bit-serial approach utilizes only serial adders and comparators. High decoding speed is still achieved because the ACS operation is carried out concurrently in each of the 2^m states. At first sight it might seem that the bit-serial solution is *pmw* times slower than the bit-parallel architecture. However, simulations at the gate level, show that serial adders and comparators have much smaller delays than their parallel counterparts, thus allowing a higher clock speed resulting in a

slow down of less than the path metric precision [Sta87]. A great advantage over all shared processor architectures is the fact that the bit-serial node-parallel architecture does not need any control logic for scheduling the ACS operation as they are performed concurrently for all states of the trellis.

3.3 Path Metric Normalization

Since only finite precision is possible for the path metric representation, the path metrics have to be normalized to prevent register overflow errors. To find the minimum number of registers to hold the path metrics, requires to determine the maximum spread Δ of the path metrics among all $2^{(K\text{-}1)}$ states [Vit79]. Assume that the partial branch metrics vary between 0 and γ. Then it follows that the maximum spread Δ for a constraint length K code is

$$\Delta = (K - 1)\nu, \tag{3.1}$$

where

$$\nu = n\gamma, \tag{3.2}$$

because any state can be reached from any other state in at most K-1 transitions. Consider the lowest path metric state a and any other state b at any node depth in the trellis, say j. There exists a path (not necessarily a surviving one) that diverged from state a $K - 1$ transitions back at time $j - K + 1$ and arrives at state b. Since all branch metrics lie between 0 and ν, the metric change to state a at time j is nonnegative, while the metric change in the path to state b must be between 0 and $(K - 1)\nu$. If the path to state b did not survive, this is only due to the fact that the accumulated metric in state b was smaller than that of this path. Hence, the spread will be even less. Thus, by subtracting the same integer from all states to bring the lowest metric to 0, the path metrics are normalized. The resulting storage requirement for the path metrics is $\lceil \log_2 (K - 1)\nu \rceil$ bits [Vit79], where $\lceil \bullet \rceil$ denotes the least integer not less than ($\bullet$).

3.3.1 Variable Shift Normalization

After a fixed number of ACS operations, the minimum path metric can be subtracted from all survivor metrics [Mag90], [Shu90]. This however, needs circuitry to find the minimum path metric and requires an additional subtracter in each ACSU. Moreover, to find the minimum metric and distribute it to all ACSUs, global communication is necessary, which usually requires much silicon area for the interconnection wires. Performing the normalization at every decoding cycle adds an additional delay for the minimum search and the subtraction.

3.3.2 Fixed Shift Normalization

Another technique of normalizing the path metrics is to provide one extra bit and delete the most significant bits (MSBs) once they are all "1" [Bre92]. This one additional bit ensures that when all MSBs are "1" there is still room to store the full range Δ of path metrics. The computational overhead is less than in the variable shift case. However, there are still global, usually long, wires.

3.3.3 Modulo Normalization

In VLSI implementations, where interconnect area can exceed that of computational area, it is crucial to keep computations local to reduce interconnect area. In [Shu90], Shung et. al. presented a modulo normalization technique performed within each ACS unit that eliminates additional circuitry and long global wires. By implementing the adders as 2's complement adders, the path metrics (PM) are normalized by *mod* C confining them to $-C/2 \leq PM < C/2$. By providing one extra bit, it is ensured that $\Delta \leq C/2$, or in other words, the path metrics are distributed only on one half of the circumference of a circle, and it is easy to determine which path metric is smaller. Let α be the angle starting from PM_1 counter clockwise to PM_2, then

$$PM_1 < PM_2 \qquad \text{if and only if} \qquad \alpha < \pi. \tag{3.3}$$

The winning metric can be determined by a 2's complement subtracter instead of a comparator. The sign bit (0 if $\alpha \leq \pi$, 1 otherwise) of the difference can be used to drive the multiplexer for the selection. Figure 3.5 shows the architecture of the ACSUs with modulo normalization. All computations are performed within the ACSU and no global communication is needed.

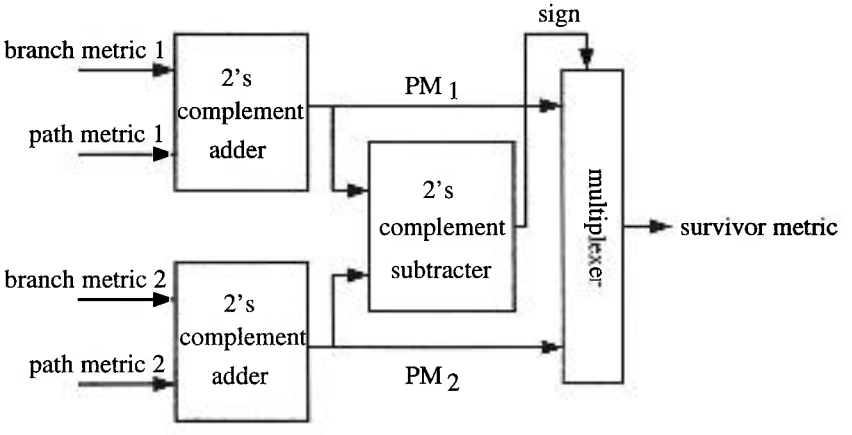


Figure 3.5 Architecture for modulo normalization with 2's complement subtracter.

However, Shung [Shu90] found that the VLSI area can still be reduced by modifying the comparison rule, because comparators can be made smaller than 2's complement subtracters:

$$PM_1 < PM_2 \quad \text{if and only if} \quad z(PM_1, PM_2) = 1, \tag{3.4}$$

where

$$z(PM_1, PM_2) = PM_{1MSB} \oplus PM_{2MSB} \oplus y\left(PM_1', PM_2'\right), \tag{3.5}$$

and $\oplus$ denotes the exclusive OR operation, the subscripts *MSB* indicate that the XOR operation is carried out only with the sign bits, $y(\bullet)$ denotes the comparison, and PM_1' and PM_2' are the path metrics without the sign bits. In short,

$$z(PM_1, PM_2) = \begin{cases} 1, & \text{if } PM_1 \leq PM_2 \\ 0, & \text{otherwise,} \end{cases} \tag{3.6}$$

$$y\left(PM_1' PM_2'\right) = \begin{cases} 1, & \text{if } PM_1 \leq PM_2' \\ 0, & \text{otherwise.} \end{cases} \tag{3.7}$$

As an example, in Fig. 3.6, PM_1 and PM_2 have opposite sign and $PM_1' < PM_2'$, giving $y(PM_1', PM_2') = 1$. Therefore, $z(PM_1, PM_2) = 0$, or $PM_2 < PM_1$. $\alpha > \pi$ confirms $PM_2 < PM_1$.

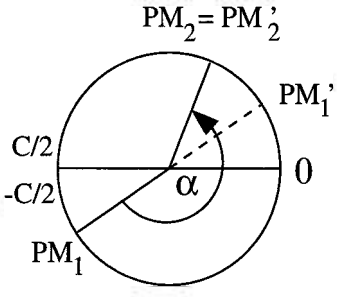


Figure 3.6 An example for the modulo normalization with the modified comparison rule.

3.4 Path Memory

The path memory stores the information sequences for each state corresponding to the surviving path metrics for determining the most likely path through the trellis. Two common techniques are in use to find the transmitted sequence [Cog89]: the *traceback method* and the *register exchange* technique.

3.4.1 Traceback Method

The traceback method requires that the decisions made in each of the 2^m ACSUs be stored in the path memory as a two dimensional array. The most likely path is estimated by stepping back one symbol of the memory at a time, starting at the most recently received information bit (Fig. 3.7). It is necessary to traceback a decision depth of d bits, where d is

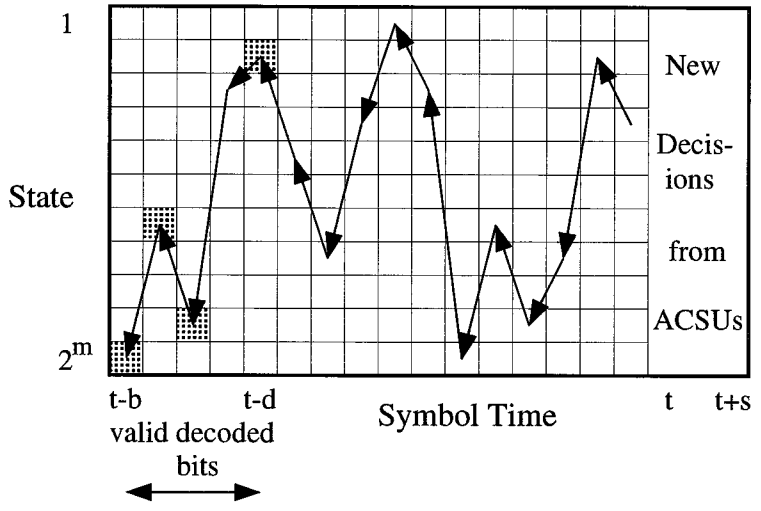


Figure 3.7 Traceback memory.

the memory length that ensures that all paths have merged with the correct one. The grey shaded boxes in Fig. 3.7 indicate $b - d$ valid decoded bits.

To understand what is stored in the memory and how the traceback works, consider the four-state trellis of Fig. 3.8. The state numbers at time k correspond to the encoder state, with a_1 and a_2 being the information bits at times $k - 1$ and $k - 2$, respectively. The new states at time $k + 1$ are represented by a_0 a_1. The discarded bit a_2 can be stored in the path memory as path information [Mag90] since it was part of the state through which the path traversed at time k.

When tracing back the trellis, the contents of the memory is read as pointers [Rad81]. The contents of a path memory cell at time $k + 1$, i. e., the path information bit a_2, is appended to the rightmost bit of the current pointer, i. e., a_0 a_1 and the leftmost bit is discarded to form a new pointer, pointing to a memory cell at time k. To illustrate the traceback method, recall the example of Fig 2.10, where only the survivor paths (Fig. 3.9) were kept. In Fig. 3.9, the branches are labelled with the path information bits stored in that node. Figure 3.10 shows the corresponding path memory. The grey shaded memory cells indicate that these

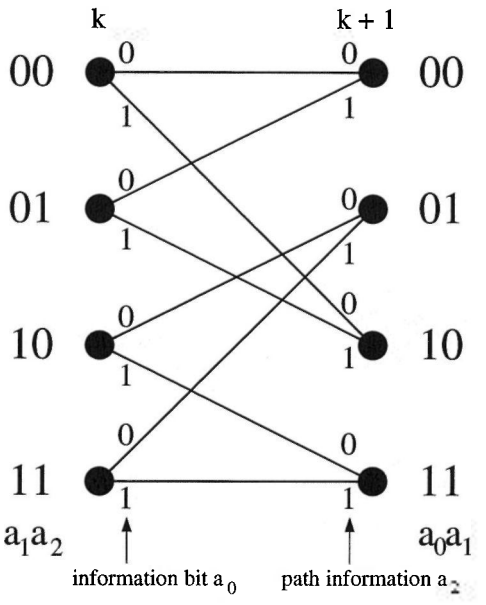


Figure 3.8 Four-state trellis to determine the path information stored in the traceback memory.

elements were read to form new pointers. Starting for example with the state with the lowest metric in Fig. 3.9, state 10, the memory cell contains a "0". The new pointer 00 is formed by appending the "0" to the rightmost bit and dropping the leftmost bit, i. e., "1", of the current pointer. One time step earlier in state 00 we find a "1", which generates the next pointer, pointing to state 01. Following this procedure back to the oldest bit, the path memory can

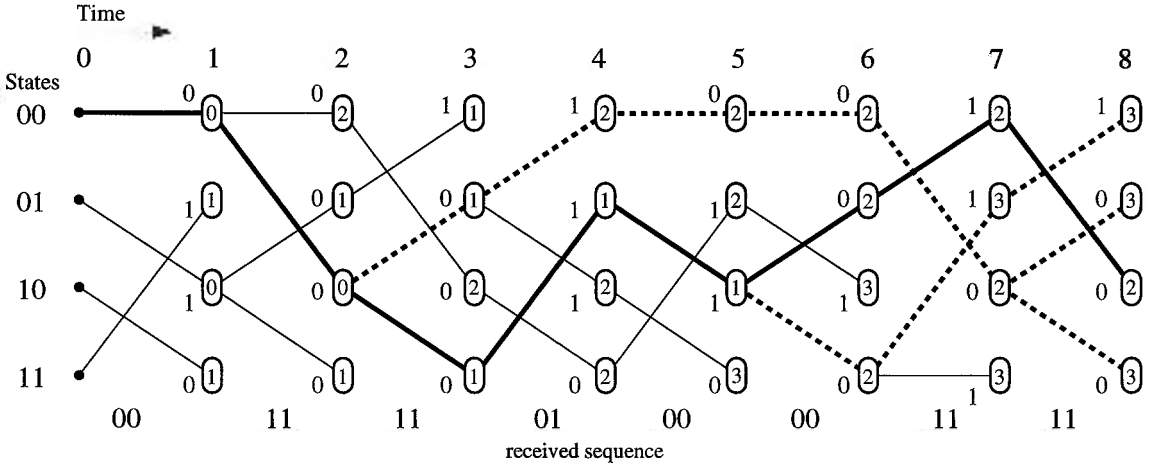


Figure 3.9 An example for the memory entry in the traceback method.

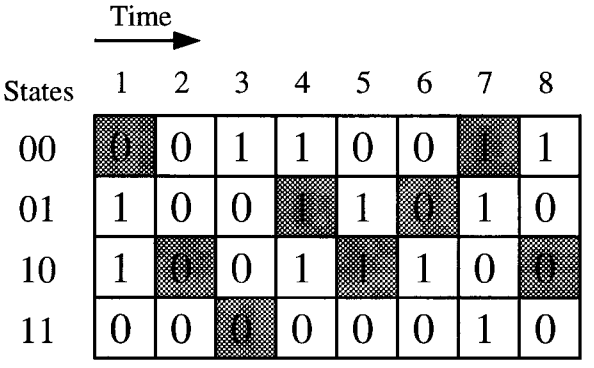


Figure 3.10 Path memory contents of the example of Fig. 3.9.

output a "0". Note, however, that the first m "decoded" bits (two in this example) are not information bits, but arbitrary bits of the state number in which the path ended.

An implementation that requires little hardware is to store the path information in a random-access-memory (RAM), configured as a circular buffer. The traceback is done by repeatedly accessing the memory and reading the path information with pointers. The pointer, the address to the RAM, is stored in a shift register. The traceback information is shifted in from the right to form the new pointer, which is the contents of the shift register read in parallel. Additional control logic is needed to control the RAM. A block diagram appears in Fig. 3.11. The major drawback of this simple implementation is its unacceptably long

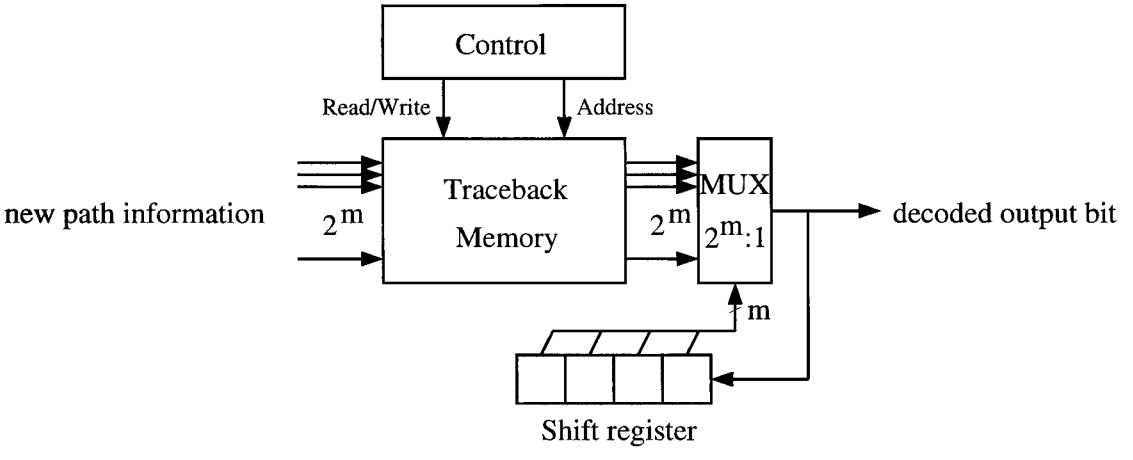


Figure 3.11 Block diagram of the basic traceback path memory.

processing time. For every new decoded information bit, the traceback method takes one WRITE cycle and d READ cycles.

A more realistic implementation is to operate the path memory with a *speed advantage* ratio of mREADs : WRITE, where $m \geq 2$ [Cog89]. For the minimum speed advantage of 2:1, the traceback performs two READs and one WRITE during one information symbol period. The minimum implementation performs $2d$ tracebacks to decode d output bits (the output bits are valid only after tracing back d steps), while further d path information bits are received from the ACSU block. The total storage requirement is $3d2^m$ bits. This storage requirement can be reduced when operating the traceback with speed advantage ratios greater than 2. In general, for any $m \geq 2$, the path memory requires the storing of $d2^m(m + 1)/(m - 1)$ bits.

3.4.2 Register Exchange Technique

The register exchange technique is based on the movement of information sequences through the path memory [Cog89] accomplished by trellis-connected shift registers one for each state. In addition to the memory elements, the register exchange technique requires multiplexers to select the surviving paths. Figure 3.12 shows an example of a four-state path

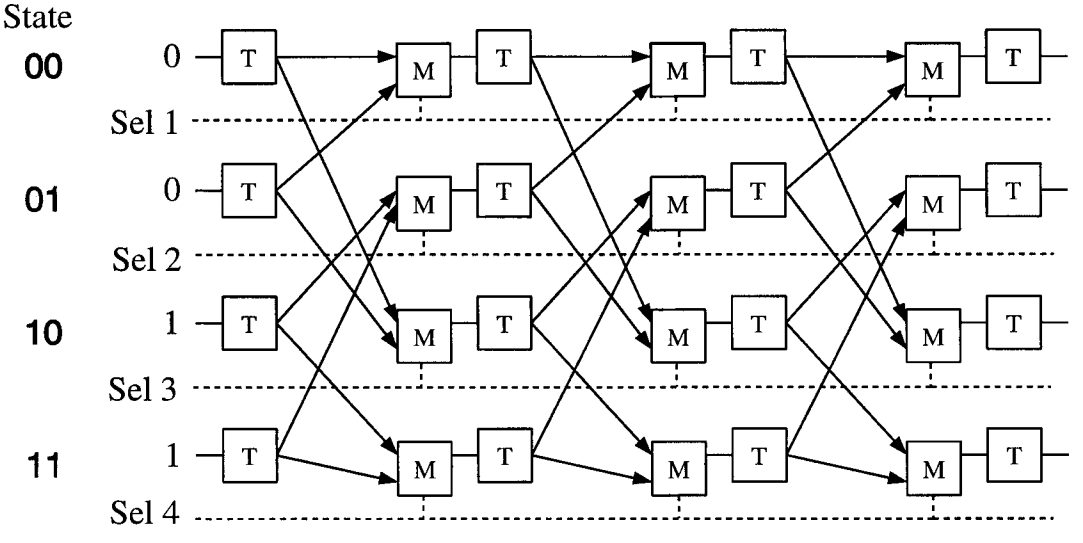


Figure 3.12 Four-state register exchange path memory.

memory, where M denotes a multiplexer and T a memory element. At each symbol clock, or in general, every time a decision in the ACS block has been made, a complete path is moved to its successor state(s) according to the trellis and a new information bit is appended to the path. For path lengths $\geq d$, the paths tend to merge and the oldest bits thus tend to represent the same information bits. The new appended information bits are readily determined. Recall from Chapter 2 that the inputs to the upper half of the trellis are all "0" because these states have an MSB of "0" indicating that the most recently encoded bit was a "0". Conversely, the states of the lower half of the trellis constantly receive "1"s as their MSBs are all "1".

Further enhancements to discard m stages of the path memory are possible as follows [Ish87]. The memory cells and their preceding multiplexers up to the m^{th} stage can be omitted, because they store information that is independent of the decisions made in the ACSUs. The improved version of Fig. 3.12 would look like Fig. 3.13 with alternating inputs of "0" and "1".

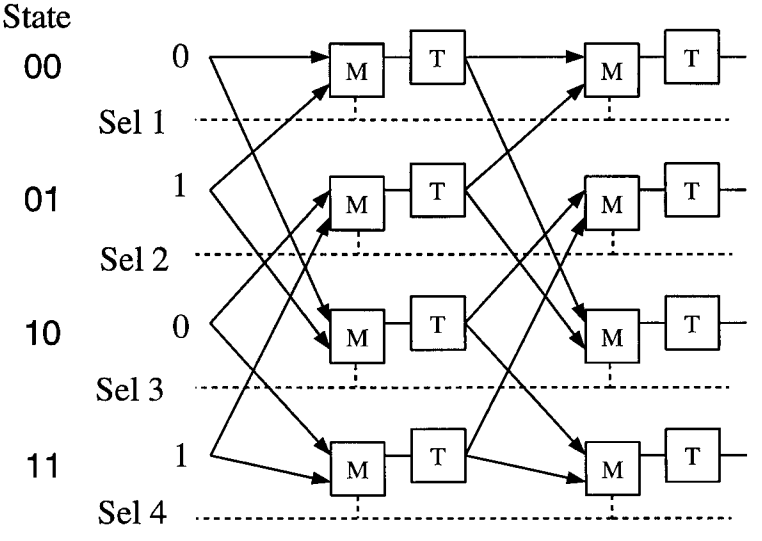


Figure 3.13 Improved register exchange path memory without unnecessary memory stages.

A major advantage of the register exchange technique is that there is no need for complicated control logic. The only required control are the select signals for the multiplexers.

These are just the decisions made by the comparator in the ACSUs. The speed of a register exchange path memory is limited only by the delay of one multiplexer and a flip-flop, making the register exchange technique attractive for high-speed Viterbi decoders.

3.5 Design Implementation Options

Before choosing a suitable decoder architecture, one has to consider the pros and contras of possible implementation technologies. This section gives a short overview of different methods of implementing a circuit. Here, we consider *programmable logic devices* (PLDs), *field programmable gate arrays* (FPGAs), and *application specific integrated circuits* (ASICs).

3.5.1 Programmable Logic Devices

A programmable logic device (PLD) is a small scale IC that can be configured by the end user to implement a specific logic function [Pel91]. A typical PLD is composed of a programmable array of logic gates and is surrounded by I/O circuitry. All different PLDs are based on variations of AND-OR plane architectures. A typical *programmable logic array* (PLA)-type structure is shown in Fig. 3.14. The X's in the AND and the OR arrays indicate a possible programmable connection. Any AND gate can be configured to implement any possible product term, as the inputs are available in their true and their complement values. The design of a PLA allows any product term in the array to be connected to any OR gate. This flexible design makes PLAs slower than other architectures with fixed connections in either the AND or the OR plane. To configure PLDs, a large number of design tools are available, e. g., CUPL, [Pel91]. These design tools accept a variety of input forms, including truth tables and state diagrams. The functions are transformed into sum-of-products Boolean form, minimized, and converted into a PLD *fuse map* [Pel91]. This fuse map is further processed into a format that can be read by the programming device.

The real benefit of PLDs comes into play when using erasable PLDs. Erasable PLDs can be reprogrammed to accommodate changes in specifications and fix design errors in a

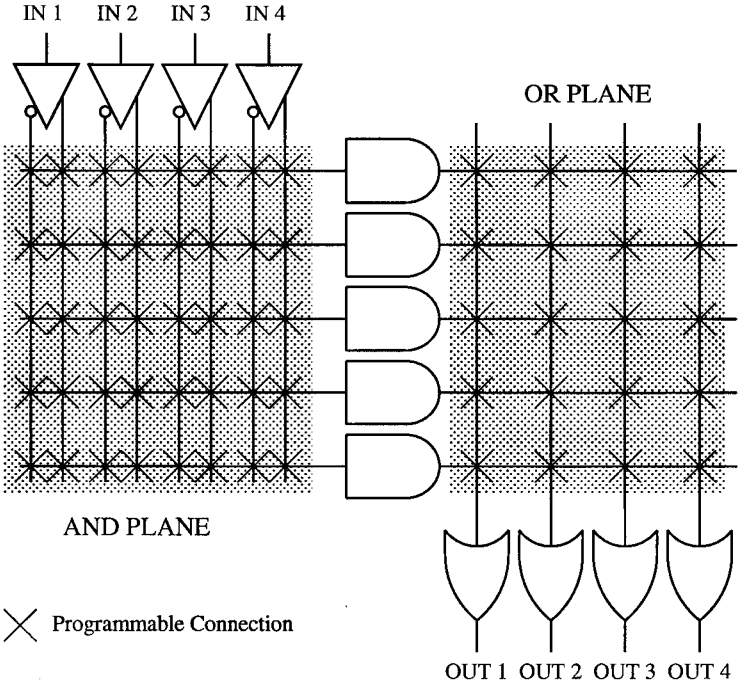


Figure 3.14 PLA-type structure of a PLD.

short time. The specifications of the design can be tested on real devices, and one does not solely need to rely on simulations. The use of PLDs is a fast way of prototyping devices. The cost of a design is in cost of the devices used.

The primary limitations of PLDs are the number of flip-flops, the number of input/output signals, and the rigidity of the AND-OR plane. The use of one function can often preclude the use of other similar functions [Xil92]. The relatively low integration level of PLDs requires many such devices for complex circuits and thus makes PLDs unsuitable for a Viterbi decoder that is to be used in mobile data communications. A constraint length $K = 5$ Viterbi decoder with decoding speed of 4.6 Mbit/s channel rate was implemented by Magerle [Mag90] using PLDs on a multi-layer board of size 25 cm by 25 cm.

3.5.2 Field Programmable Gate Arrays

A field programmable gate array (FPGA) is a high-density programmable device with more functionality than a PLD that can be configured by the end user [Xil92]. An FPGA consists of a matrix of independent logic modules in the interior of the device and a ring of I/O blocks that can be connected to form a larger circuit. Interconnect resources occupy the channels between rows and columns of the logic blocks. Several different FPGA families with different architectures exist at present. Since the design tools available at the University of British Columbia are specifically for Xilinx's FPGAs, the following description will be restricted to those devices. A good overview of other FPGA families can be found in [Pel91], [Ros93]. Figure 3.15 shows the structure of Xilinx's Logic Cell™ Array (LCA™).

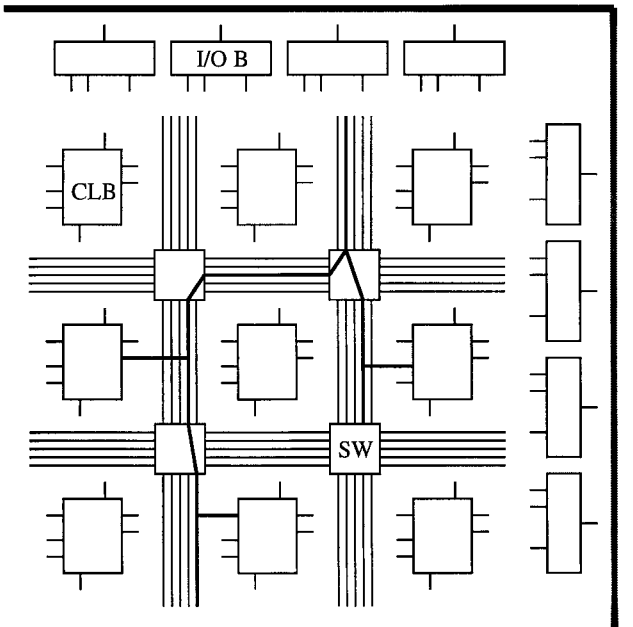


Figure 3.15 FPGA architecture.

Each configurable logic block (CLB) is capable of implementing any Boolean function of its inputs. The basis of a CLB is a small static random access memory (SRAM), functioning

as a lookup table [Ros93], a flip-flop, and feedback logic. The I/O blocks can be configured as either inputs or outputs or bidirectional. The routing architecture consists of horizontal and vertical routing channels and switch blocks at cross-over points of routing channels. The switch blocks or *switch matrices* can switch signals from one path to another. Design tools, such as Xilinx's XACT take schematics or behavioural description as design entry and create netlists of the circuit. A logic optimizer reduces Boolean equations and partitions large combinational parts to fit the size of the CLBs. Then, the individual CLBs are placed, i. e., the logic is assigned to certain CLBs, and routed. The final step in designing FPGAs is to create a bit pattern that configures the actual FPGA. The bit pattern is either directly loaded into the FPGA RAM or can be stored in a configuration programmable read only memory (PROM) on the board.

Xilinx's FPGAs are programmed on static RAM technology. A drawback of SRAM programming is that the configuration pattern is volatile, i. e., every time the system is powered up, the configuration pattern has to be loaded from an external PROM. However, since the chip programming is done with memory cells, the FPGA can be programmed an unlimited number of times. Prototyping and in-circuit verification can replace extensive simulations. A designer can verify that the design works in a real system and does not have to rely merely on potentially-erroneous simulation models of the system. Any design improvement can be accomplished within a few hours. Since the FPGAs rely solely on programming a certain configuration, there is no fixed cost for expensive mask production and again cost is cost of the device.

A big drawback of FPGAs is that though high toggle rates (125 MHz) [Xil92] of flip-flops are claimed by Xilinx, the system clock rate is about one third to one half the maximum toggle rate. The switch matrices consist of pass transistors connecting or not two wire segments, depending on the value in the controlling memory cell [Tri93]. This pass transistor introduces

resistance into the routing path and hence delay. The delay is strongly dependent on the number of interconnect points a signal path is passing and cannot be determined by logic simulations during design, because the simulator has not placed and routed the circuit yet.

A $K = 5$ Viterbi can probably be implemented on a single FPGA, plus a RAM for the path memory and a PROM to configure the FPGA upon power-up. An FPGA implementation is certainly interesting, especially for prototyping and proof of concepts. Unfortunately, FPGA design tools were not available at the University during design phase of the Viterbi decoder, therefore this option had to be dropped.

3.5.3 Application Specific Integrated Circuit

The highest level of integration can be achieved with application specific integrated circuits (ASICs). Two types of ASICs exist on the market today: *Mask-Programmed Gate Arrays* and *Standard Cell and Custom ICs*. In typical standard cell implementations, as can be seen in Fig. 3.16, the standard cells are placed in rows across the chip, leaving horizontal

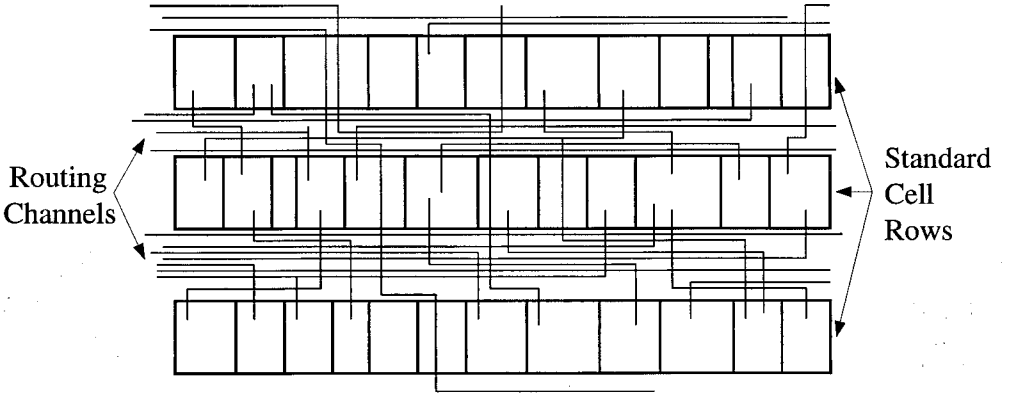


Figure 3.16 Typical standard cell ASIC structure.

channels for routing. Wire connections are only made when necessary, resulting in different routing channel widths. Vertical routing is mainly accomplished by vertical wires across the standard cells and vertical channel at either end of the standard cell rows. The design tools typically accept schematic or higher level language entries and perform the complete

chip layout very quickly. If further area reductions deem necessary, time consuming custom layout blocks can be included into the design and routed with the standard cells. The layout is then sent to an ASIC manufacturer, called a foundry, for production. The need for unique masks for all layers used in manufacturing imposes high costs and weeks or months of delay for development. This requires careful circuit simulations before mask production, as the design cannot be changed afterwards. For high volume applications (> 100,000), standard cell and custom ICs result in the lowest production costs [Xil92].

Gate arrays implement user logic by interconnecting transistors or simple gates during the last stages of manufacturing process. Unlike standard cell ICs, mask-programmed gate arrays costs include fixed costs for mask production as well as cost per unit. Gate arrays become cost effective for volumes around 100,000.

With the support of the Canadian Microelectronics Corporation it was possible to access Northern Telecom's 1.2μm CMOS process, making a custom IC a viable option. The K = 5 Viterbi decoder was implemented on a single chip, helping to reduce size and weight of mobile communications devices.

4. Design of the Variable-Rate Viterbi Decoder

4.1 General Considerations

The Viterbi decoder is designed to decode convolutional codes of constraint length K = 5. Figure 4.1 shows the encoder of memory m = 4, generating the convolutional code. The

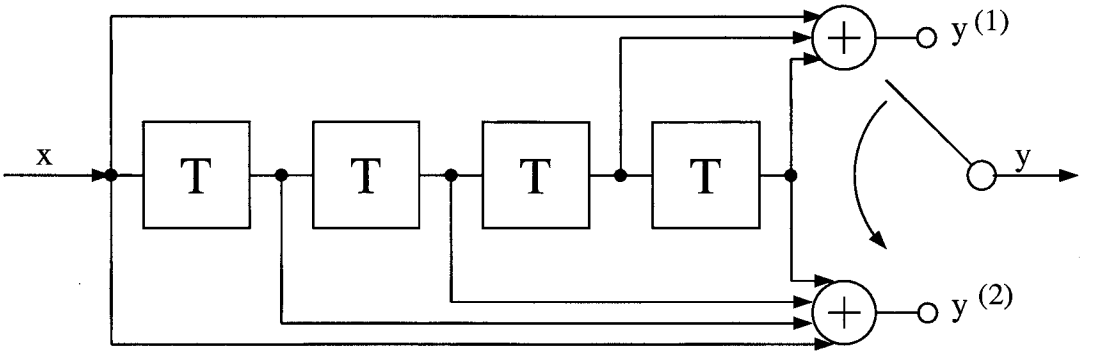


Figure 4.1 Convolutional encoder

decoder is capable of decoding any code of rate ranging from $(V-1)/4(V-1)$ to $(V-1)/V$ for $1 \leq V \leq 8$, or any code rate from 1/4 to 7/8 with period $P \leq 7$. The mother 1/2 rate code, from which all other codes are derived, has a free distance d_{free} = 7. For the highest code rate of 7/8, the code has still a d_{free} = 3. For high coding gain, the decoder expects eight-level quantized channel symbols as its inputs. The assumed modulation schemes are BPSK and QPSK. Since available silicon area was more a concern here than was decoding speed, the bit-serial node-parallel architecture is the most attractive and therefore adopted. The rest of the chapter describes specific implementation issues of the functional blocks used in the variable-rate Viterbi decoder.

4.2 Branch Metric Unit

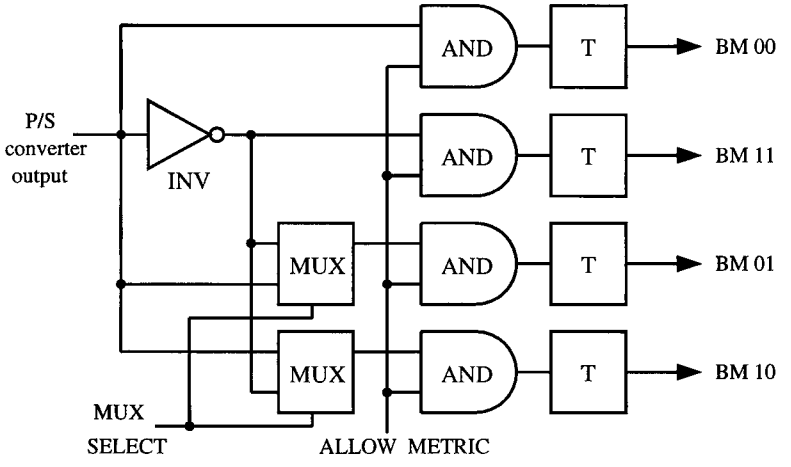


Figure 4.2 Implementation of the bit-serial branch metric unit.

The bit-serial branch metric unit (BMU) for BPSK is a very simple block, as shown in Fig. 4.2. It takes the 3-bit channel symbols that have been converted to a serial bit stream by the parallel-to-serial converter, arriving with the least significant bit (LSB) first, and compares the individual bits against "0" and "1". For the QPSK case, I and Q channel symbols are fed in serially. Figure 4.3 illustrates the operation. Without loss of generality we assume a simple 1/2 rate decoder. The only difference with variable-rate codes is a different MUX SELECT signal from the control block for different code rates. Note that the branch metrics calculated here are only partial branch metrics for each channel symbol, having a range from 0 to 7. The first row in Fig. 4.3 shows the output of the parallel-to-serial converter. The bit clock is illustrated in the second row. The LSB of the serial channel symbol arrives at every eighth clock cycle, which follows from the required path metric precision as discussed in the next section. Recalling a trellis, let the branch metrics for the four expected channel symbols "00", "01", "10", and "11" be BM 00, BM 01, BM 10, and BM 11. From Table 3.1, the metric for an expected "0" is always the same as the channel output, while for an expected "1" it is its 1's complement (the bitwise inverse). Therefore, BM 00 is just the

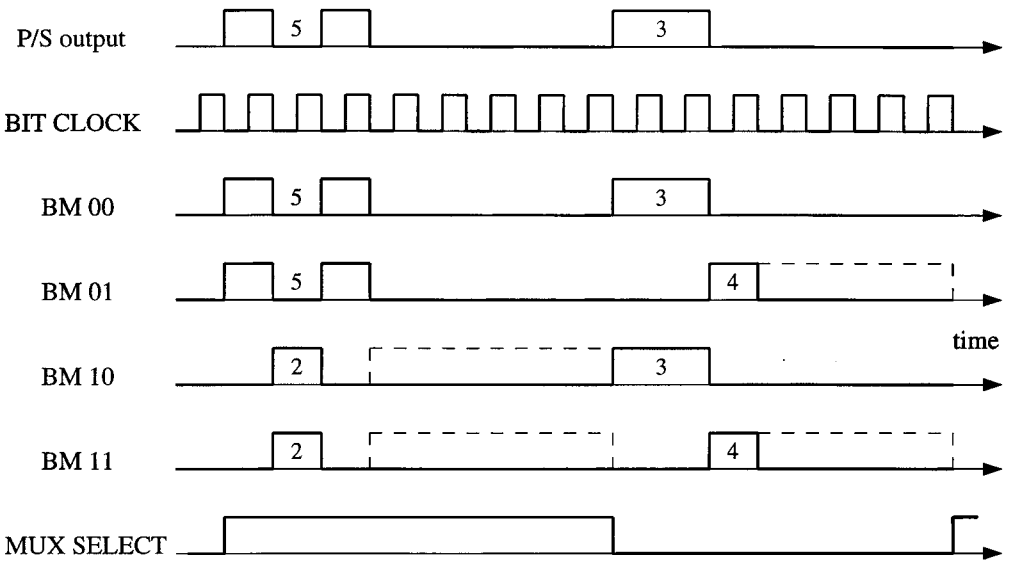


Figure 4.3 An example of the branch metric calculation.

output of the parallel-to-serial converter and BM 11 is its inverse. In the case of BM 01 and BM 10, BM 00 and BM 11 are tapped, respectively, for the first symbol and then switch the multiplexer MUX to tap their complements. The control block provides the "MUX SELECT" signal, which of course is adapted to the code matrix for different codes. The AND gates are necessary because the complement of the parallel-to-serial converter would also give a logic "1" between two symbols, as the dashed lines in Fig. 4.3 indicate, and the ACSU would interpret the resulting output as large branch metrics. The signal "ALLOW METRIC" is "1" only while the three converter bits are shifted out. A pipeline stage between the BMU and the ACSU using the memory elements T shortens the critical path in the ACSU. Minor area improvement and speed-up is achieved by using NAND instead of AND gates, and performing the necessary buffering of the branch metrics with faster inverters.

4.3 Add-Compare-Select Unit

The add-compare-select unit (ACSU) adds the new arriving branch metrics to the current path metrics, compares the updated path metrics, and keeps the smaller path metric as

survivor. In order to store the required path-metric precision, it is necessary to determine the maximum spread Δ among all path metrics. From Eqn. 3.1 it follows that $\Delta = (K - 1)\nu$. The maximum branch metric occurs in the case of the rate 1/4 code, because one information bit generates four channel bits in the encoder. Therefore the partial branch metrics of each of four channel symbols are added in the decoder before making a decision. The maximum branch metric ν = 4 x 7 = 28. Thus, following Eqn. 3.1, Δ = 4 x 28 = 112, which can be stored using 7 bits. Providing one extra bit precision and using the modulo normalization with the modified comparison rule appears to be the most area efficient normalization and is therefore adopted in this design.

For a bit-serial implementation of the ACS operation, the following observations can be made [Sta87]:

1) a serial adder starts with the least significant bit (LSB) first;
2) a serial comparison is fastest starting with the most significant bit (MSB);
3) a select operation is possible only after a completed comparison.

Obviously, Observations 1) and 2) are conflicting and suggest that we have to reverse the order of the bits in a first-in last-out (FILO) register. Observation 2) and Observation 3) imply a buffer, thus increasing the operation's delay by another word length, i. e., eight clock cycles in our case. However, using a serial comparator that starts with the LSB allows to pipeline the add and compare operation. The bit-serial 3-bit quantized branch metrics (BM) enter the ACS cells with the LSBs first and are added in the 2's complement adders ADD to the current path metrics (PM) (Fig. 4.4). Both sums, i. e., the updated path metrics, are stored in shift registers, denoted by memory elements T in Fig. 4.4, and fed to the comparator COMP concurrently. The result for every bit of the comparison is stored in the flip-flop FF, which is clocked at every symbol period T_s after the MSBs have been processed. The carry bit of the bit-serial comparison does not have to be cleared after completed comparison

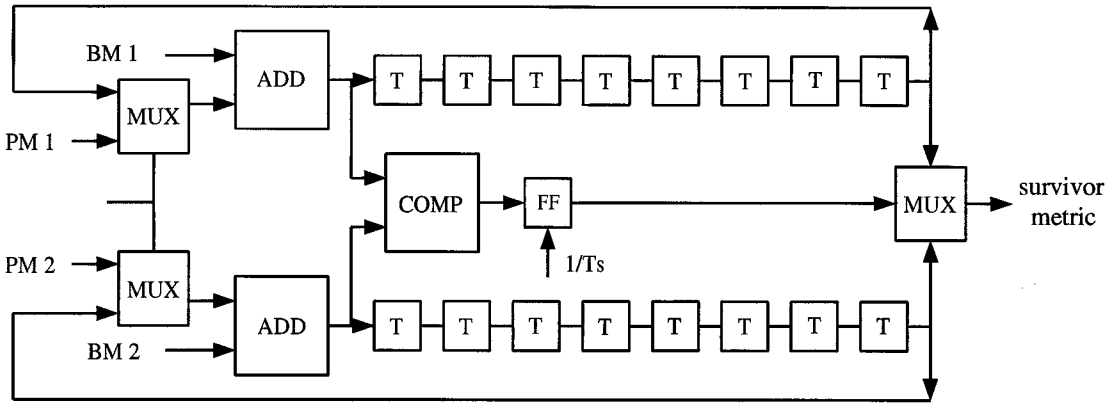


Figure 4.4 Block diagram of a bit-serial ACS unit.

because in the case of a tie, both shift registers store the same value, and it does not matter which one is selected. The multiplexer to the right in Fig. 4.4 selects the smaller sum as the survivor metric. The carry bits of the 2's complement adders have to be reset after the MSBs have been processed to avoid overflow and invalidation of the following addition. Since negative edge-triggered flip-flops are used, the same control signal can be used for triggering the selection-flip-flop FF and injecting a "0" to the adder, thus avoiding an extra global signal for carry reset. This control signal FF CLOCK-CARRY RESET is "0" during every arrival of an LSB at the adder to inject the "0", but at the same time its falling edge triggers the flip-flop FF after the previous MSB has been processed and the comparison is finished. Figure 4.5 shows the timing diagram to reset the carry bits in the adders and trigger the selection-flip-flop FF. The third row in Fig. 4.5 displays the positions of the

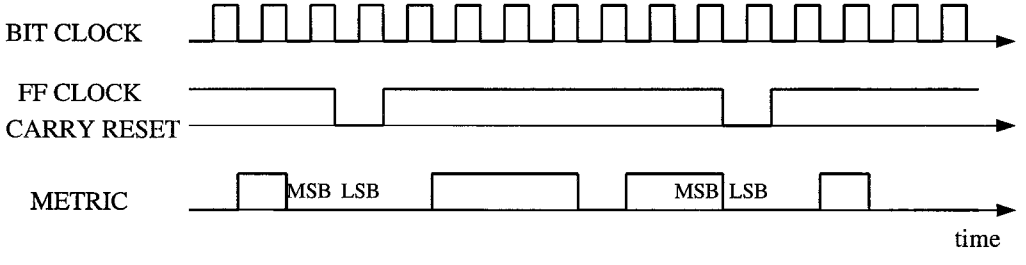


Figure 4.5 Timing diagram for the concurrent carry reset and flip-flop FF trigger.

MSB and LSB of some example metric. In the variable-rate decoder, described here, up to four partial branch metrics have to be added to the current path metrics before a decision about the survivor can be made. Therefore, the left multiplexers in Fig. 4.4 feed back the intermediate sums and accept path metrics from predecessor states only for the first symbol of an information bit after a decision has been made. Otherwise, the "survivor metric", which is output continuously, represents invalid data and has to be blocked by the multiplexers.

4.3.1 Layout of Add-Compare-Select Units

Since the concern here is silicon area, custom layout for the ACSUs was considered necessary. This section describes the individual gates necessary to build an ACSU.

Multiplexers were laid out, because a standard cell static multiplexer uses 12 transistors and is designed to drive bigger capacitances than is actually needed here. A smaller solution is a six-transistor CMOS transmission-gate multiplexer. A CMOS transmission gate (shown in Fig. 4.6) is an "ON-OFF" switch consisting of an NMOS and a PMOS transistor in

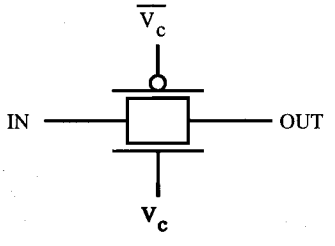


Figure 4.6 CMOS transmission gate.

parallel [Gei90], i.e., drains connected to each other, and sources connected to each other. If the control voltage V_c = 0 V, the gate voltage of the n-channel transistor is also 0 V, while the gate voltage of the p-channel transistor is V_{DD}, both transistors will be "OFF" and OUT will assume a high impedance state. In the case $V_c = V_{DD}$, both transistors of the transmission gate will conduct. The NMOS transistor pulls the output to 0 V, if IN is at 0V, and the PMOS transistor pulls the output to V_{DD} if IN is at V_{DD}.

The multiplexer consists of two transmission gates and an inverter to provide the inverse of the select signal. Figure 4.7 shows a gate level representation of the multiplexer. A plot of the layout and the multiplexer's characteristics are summarized in Appendix A. SB

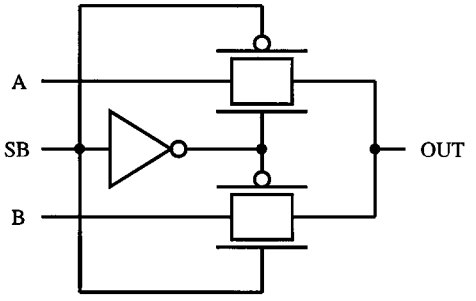


Figure 4.7 Transmission gate multiplexer.

("Select B") is the control signal that selects which transmission gate should be conducting and therefore which input should be connected to the output. For SB = 0 V, the upper transmission gate is "ON", thus connecting A with the output OUT. The lower transmission gate is "OFF", isolating B from the output. If SB = V_{DD}, B is selected and the upper transmission gate is "OFF" isolating A. Minimum-size transistors were used in the layout. A minimum-size transistor has both the channel length l and width w set to the allowed minimum for the technology used. For the 1.2 μm technology a minimum-size transistor has l = 1.2 μm and w = 2 μm.

From the block diagram of the ACSU, Fig. 4.4, the path metrics are stored in memory elements arranged as shift registers. This suggests the use of D-flip-flops to transfer the signal at their inputs to the outputs at every falling clock edge. For the flip-flop implementation, several possibilities can be found in the literature, e. g., [Jir87], [Cha89]. To keep the silicon area small, static flip-flops similar to the standard cell approach which need 30 transistors were precluded. The single-phase-clock dynamic D-flip-flop in [Jir87] uses ten transistors. Unfortunately, the diffusion area of such flip-flops cannot be shared efficiently, so the area saving is relatively small compared to the standard cell. A better solution arises with the

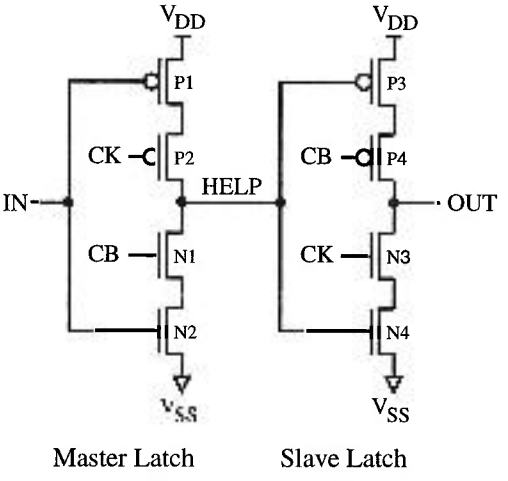


Figure 4.8 Structure of the "clocked inverter" dynamic *D*-Master-Slave flip-flop.

use of two-phase clocks, CK and CB and the use of two clocked inverters [Cha89], to build a master-slave flip-flop. Figure 4.8 shows such a structure. The outer four transistors build two inverters (P1-N2, P3-N4), which are never transparent simultaneously, provided the clock phases are as shown in Fig. 4.9. CK is the chip clock, and CB is derived from CK by an inverter. The clock skew $|t_2 - t_1|$ between CK and CB is introduced by the inverter through which CB is derived from CK. To describe the flip-flop's operation, it is necessary

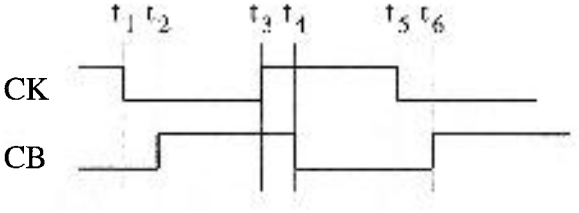


Figure 4.9 Clock traces for a dynamic *D*-Master-Slave flip-flop.

to consider the case where *IN* is "1" and "0" separately, because they lead to different results. In the following $t_i \rightarrow t_j$ symbolizes the time frame from t_i to t_j.

1. *IN* = "0":

$t_1 \rightarrow t_3$ Master latch precharge phase.

$t_3 \rightarrow t_5$ Slave latch evaluation phase. OUT = 0.

2. *IN* = "1":

$t_2 \rightarrow t_4$ Master latch precharge phase.

$t_4 \rightarrow t_6$ Slave latch evaluation phase. OUT = 1.

All n-channel transistors are 3.1μm wide. Figure 4.10 shows that 3.1μm wide transistors require less cell height and are therefore better suited when the transistors are stacked like in Fig. 4.8. A minimum size transistor at the input can pull the input line to "0" at the start-up phase and reset the path metrics.

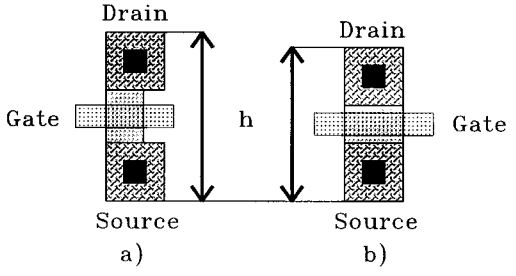


Figure 4.10 a) Minimum-size transistor with channel width w = 2μm, b) transistor to achieve minimum cell height with w = 3.1μm.

While the flip-flop works fine when succeeded by a static gate, HSPICE^{TM} simulations of a shift register showed that the signal deteriorates after three flip-flops, as shown in Fig. 4.11 for a time frame of 25ns, and yields a logic error after the third flip-flop. The first two rows in Fig. 4.11 show CK and CB, respectively, and the fourth and sixth row show the flip-flops' intermediate nodes, denoted HELP in Fig. 4.8, of the second and the third flip-flop. Rows three, five, and seven are the outputs of the first, second, and the third flip-flop. In any fabrication process, drain and source will overlap the gate and form what can be modelled as two capacitors, as shown in Fig. 4.12 for a PMOS transistor. These parasitic *overlap capacitances* induce voltage into a floating node when the gate of the transistor is switched. A changed charge can partly turn on transistors. The effect accumulates over several stages until the output yield a logic error. In Fig. 4.11, the output of the first flip-flip is floating up to time t = 35ns, since CK = "0" and CB = "1". At time t = 35ns, when CK goes to

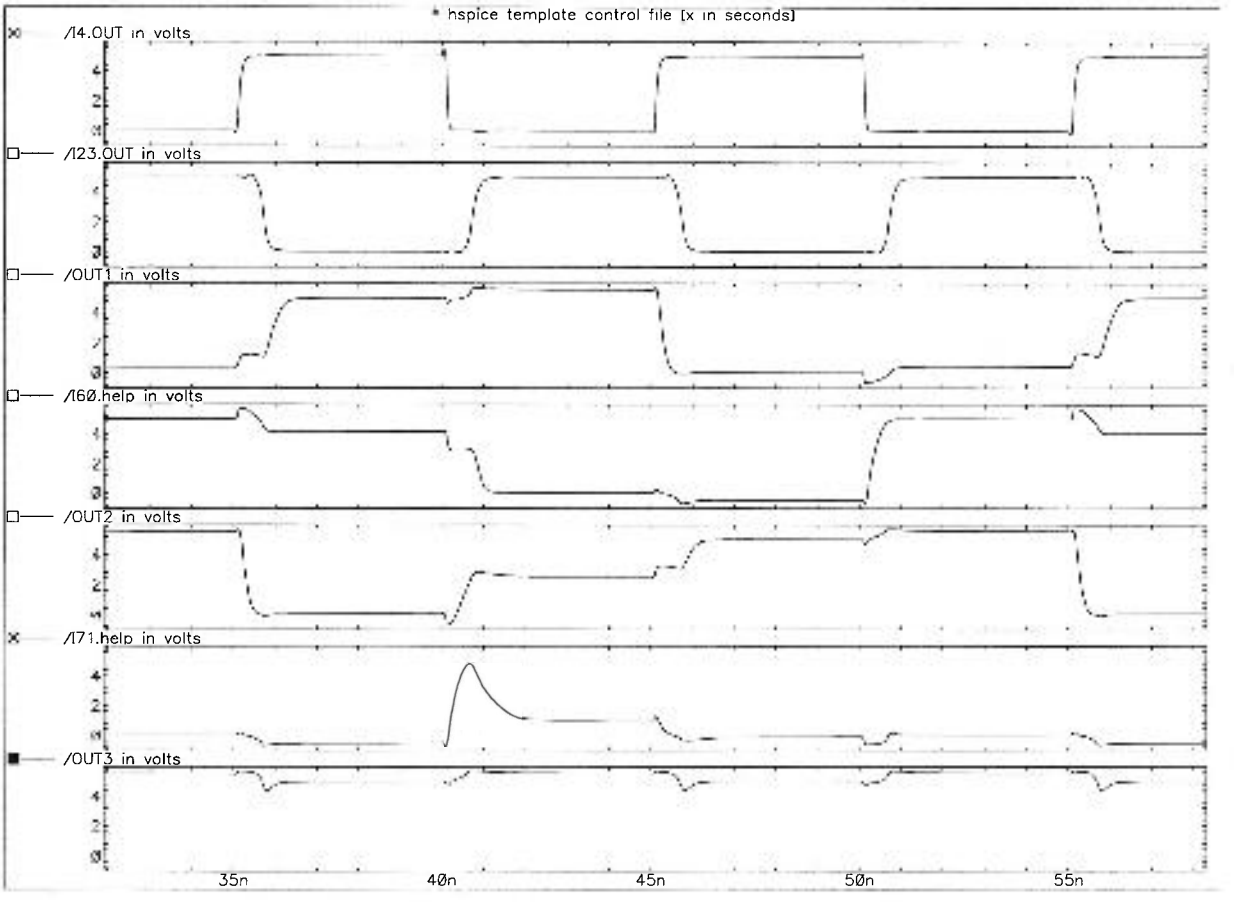


Figure 4.11 Flip-flop waveforms with clock skew = 0.7 ns.

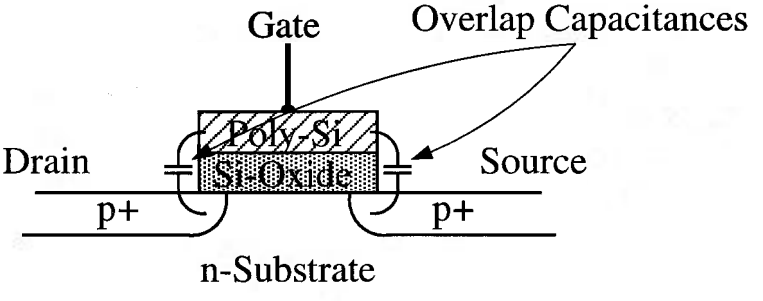


Figure 4.12 Cross section of PMOS transistor with overlap capacitances.

"1", the overlap capacitance between CK and the output in transistor P4 pulls up the floating output. The voltage of the output is high enough to turn on N2 of the second stage slightly and create a path to V_{SS} while both clocks are "1". With CK = "1", P2 is turned off, the

intermediate node in the second stage becomes a floating node and will be pulled down by the path to V_{SS} to some value less than "1". At t = 40ns, CK goes to "0" pulling down the floating intermediate node even lower via the overlap capacitance between CK and the intermediate node in transistor P2. The "non-one" intermediate node turns on transistor P3, while both clocks are "0". At the same time, the output becomes floating and is charged up by the path to V_{DD}, formed by the transistors P3 and P4. Figure 4.11 shows that the effect aggravates from stage to stage until the floating intermediate node of the third flip-flop is discharged while both clocks are "0" and the output of the third stage outputs a constant "1".

For clock skews $t_2 - t_1 < 0.3$ns, the time both clocks "0" or "1" is too short to charge floating nodes enough to turn on transistors. However, it is not possible to guarantee a clock skew < 0.3ns when using two big clock trees (one for CK and one for CB) to drive 3200 flip-flops. A solution is to increase the width of the PMOS transistor of the inverter in the master latch to at least 13μm. The improvement can be clearly seen in Fig. 4.13 with a transistor width for P1 of 13μm. This increase in transistor size increases the capacitive load on the previous flip-flop and the output will not be charged high by overlap capacitances, thus allowing clock skews up to 0.6ns. The complete layout of the resetable D-flip-flop can be seen in Appendix A.

Two XOR gates are needed for the normalization to perform the comparison. An attractive solution is to use a transmission gate implementation [Wes88] with only 6 transistors, as shown in Fig. 4.14. The operation of such an XOR gate is as follows.

When A is "1", transistor 2 is "ON" and the source of transistor 4 is pulled to V_{SS}, thus transistor 3 and 4 can act as an inverter. The transmission gate is open and not conducting. The output of $\overline{B}$ by the inverter forms A$\oplus$B. In the case A is "0", transistor 1 pulls the source of transistor 4 to V_{DD} and disables transistor pair 3 and 4. The transmission gate is now closed, passing B to the output. The layout and a summary of the XOR's characteristics are found in Appendix A.

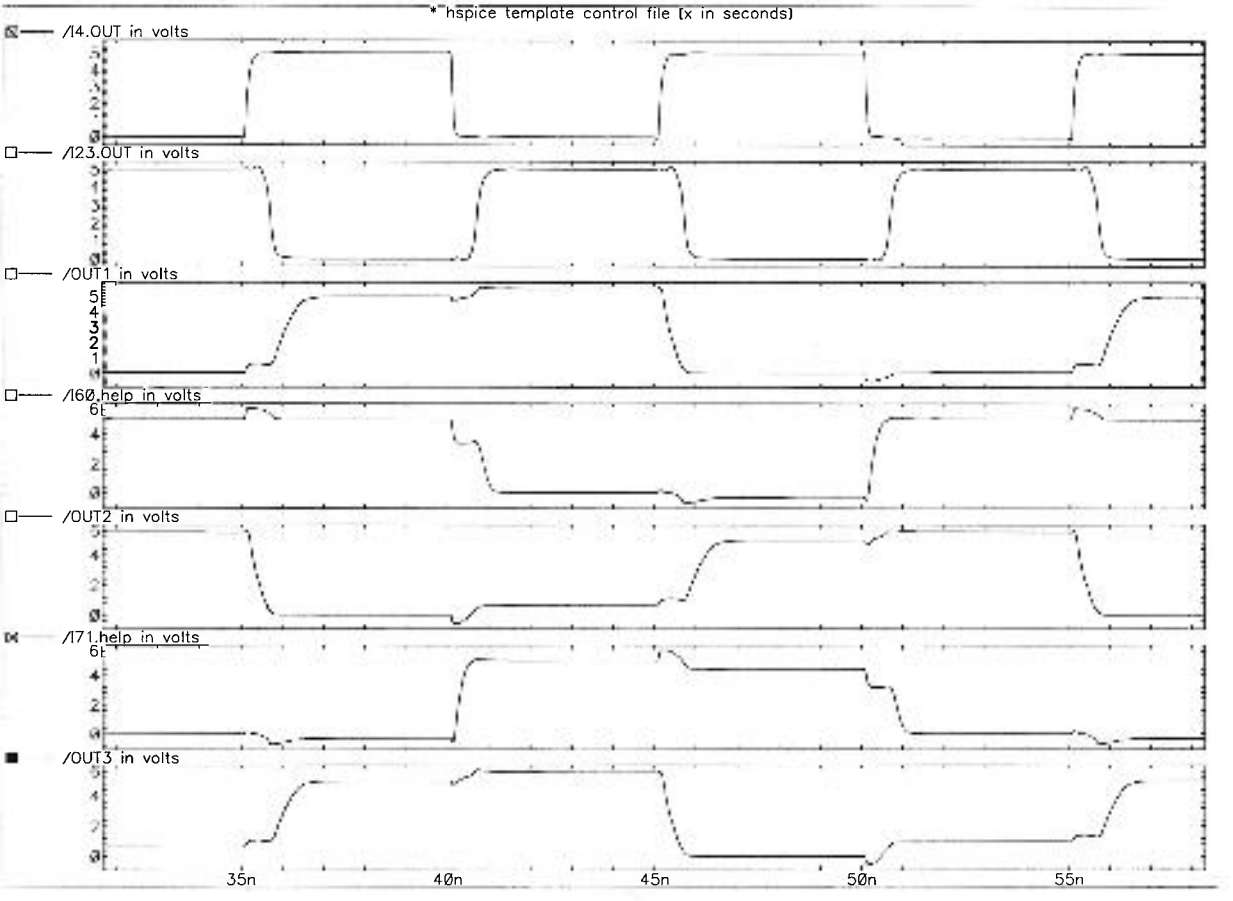


Figure 4.13 Waveforms of the improved D-flip-flop with transistor width of P1 of 13μm.

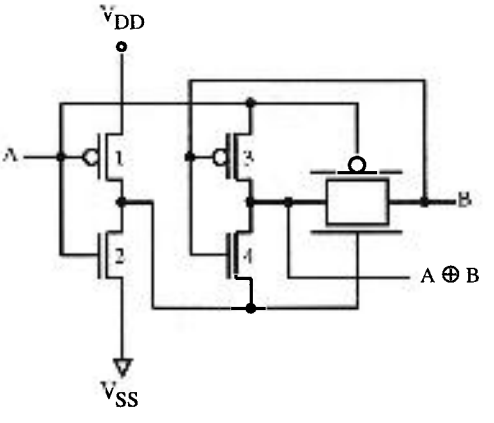


Figure 4.14 Transmission gate exclusive-or gate.

The layout of the adders is a cell-height reduced version of the layout found in [Wes88].

The comparator is implemented as the carry stage of an adder, receiving one input inverted. Since the layout of the adder is split into the sum stage and the carry stage, the layout of the carry stage can by directly used as comparator.

After finishing the layout of a complete ACSU, simulation with HSPICE^{TM} showed that the critical path through the multiplexers, the adder, and the comparator is 8ns. Allowing a safety margin of 2ns, the maximum clock frequency is limited at 100 MHz.

4.3.2 Pairing of Add-Compare-Select-Units

After completing the layout of the ACSU, the ACSUs were paired to reduce the number of interconnect wires by two [Bre92]. The trellis is a regular structure that can be divided into a set of butterflies. In any trellis diagram for $1/n$ codes, an ACSU receives two inputs from ACSUs whose state number differs only in the LSB. In the other half of the trellis, there is an ACSU whose state number differs only in the MSB that receives the same inputs. By pairing those two ACSUs, their inputs can be shared, thus reducing the interconnect wires by two. Figure 4.15 shows a four-state example of how the pairing of the ACSUs can save interconnect area. In Fig. 4.15a, the ACSUs, labelled by the binary representation of their corresponding state, are connected in a conventional trellis-like manner. The wiring in Fig. 4.15a needs four vertical wiring tracks, while the paired ACSUs in Fig. 4.15b need only two vertical tracks. Expanding this to the 16-state Viterbi decoder described here, implies that instead of 16 vertical tracks, the butterfly-paired ACSUs need only eight vertical tracks. The complete wired ACSU block for the 16 state decoder needs 1.5 mm^2 of silicon area.

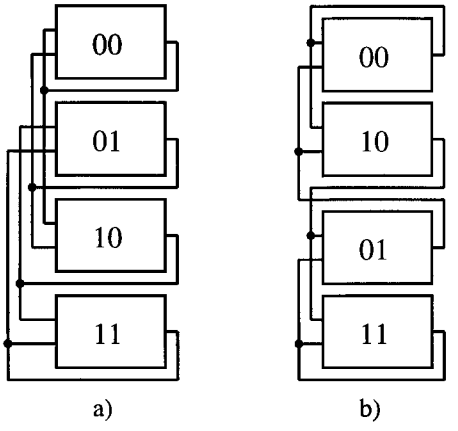


Figure 4.15 a) Conventional ACSU wiring, b) ACSU pairing that saves half the interconnect wiring.

4.4 Path Memory with Novel Area-Saving Layout

The path memory has to store the information sequences for each state corresponding to the surviving path metrics. To determine the most likely path through the trellis, which represents the transmitted sequence, two common techniques are in use [Cog89]: the *traceback method* and the *register exchange* technique.

Simulations of a RAM macro cell show a READ delay of 10 ns and a WRITE delay of 5.3 ns. Allocating some time for the control logic and a safety margin of about 2 ns, 20 ns is required for READs and WRITEs as the next multiple of the bit duration of 10 ns. This would allow a speed advantage ratio of 3:1 in a 80 ns symbol period. The memory length now becomes $1.7d$. In [Yas83] it was found that a suitable decoding depth d for the limiting 7/8 rate code is around 14 times the constraint length, i. e., 70 for the $K = 5$ decoder here. As the RAM cell generator of the VLSI tool Edge^{TM} creates RAM of sizes of powers of 2, a RAM of 16 x 128 bits is necessary, which needs 3 mm^2 of Si area. Though a RAM layout is very compact, a major cost of the traceback method is its considerable amount of control for address generation, READ/WRITE clocks, bit selection, and output bit buffering.

The reason for choosing the register exchange technique is based on a comparison of the two methods in [Cog89]. The authors of [Cog89] found that a 64-state path memory

implemented in trace back with a speed advantage of 6:1 occupies the same Si area as a register exchange path memory, implemented as a full custom layout block. In a 16-state Viterbi decoder, the amount of control logic will remain basically the same as for a 64-state decoder and the storage requirement for a 3:1 speed advantage is greater than for 6:1. On the other hand, the size of the register exchange memory decreases proportionally with the number of storage cells. Since the speed of either technique was sufficient for the given requirements, the smaller solution was adopted.

The decoded output bit is chosen by a majority vote of five outputs. The number of gates increases very rapidly with more than five outputs, while the five output majority vote can be implemented efficiently with mostly three-input NOR gates from the standard-cell library, as it chooses any three of five inputs.

To keep the circuit area small, the path memory cells are implemented using 6-transistor transmission-gate multiplexers and 6-transistor dynamic latches with NMOS pass transistors, similar to the ones used in [Ish87]. This is illustrated in Fig. 4.16. CB and CK are two

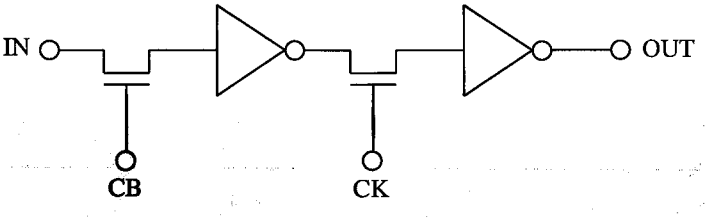


Figure 4.16 6–transistor dynamic latch.

non-overlapping clocks, as shown in Fig. 4.17, generated by the control circuit, to avoid transparency of the latch and hence races. The use of NMOS pass transistors limits the lower

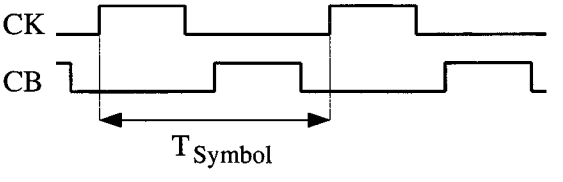


Figure 4.17 Two non-overlapping clocks CB and CK for the path memory.

speed of operation as the transistor capacitance can hold its voltage value only for a certain amount of time. Simulations showed that this lower speed limit is around an information rate of 300 bit/s, which is far below most transmission speeds. Unfortunately, the pass transistors show some DC voltage loss at their outputs due to the threshold voltage and no pull-up transistor to set the output of the pass transistor to V_{DD}. Although that results in a reduced noise margin, NMOS pass transistors were used instead of the more robust but 30% bigger full CMOS transmission gates because of the very limited silicon area available. The price for the small silicon area is some static power dissipation of the flip-flops. If the output of the pass transistors is a threshold voltage below V_{DD}, the lower voltage slightly turns on the PMOS transistor of the succeeding inverter and creates a DC path from V_{DD} to ground. The complete layout and speed characteristics of a path memory cell, consisting of a transmission-gate multiplexer and a flip-flop, can be found in Appendix A.

A major part of the silicon area used for the register exchange technique results from the trellis wiring between consecutive stages. A straightforward implementation of the trellis wiring would require 16 vertical wiring tracks for every stage, as shown in Fig 4.18a. The total wiring area is about 25% bigger than the total memory cell area. Using a state relabelling technique presented in [Cog88], [Cog89] can reduce the interconnect area substantially. Coggins partitions the 2^m state trellis into 2^{m-r} trellises with 2^r states each. The movement of the surviving paths is confined to a particular region for r stages. Every r^{th} stage relabelling wires move the paths to the appropriate region. Coggins' technique applied to a 16 state trellis would partition the trellis into four four-state trellises and use a relabelling every second stage. The total number of vertical wiring tracks is 16 for two stages, thus reducing wiring area by 50%.

We propose an approach, often used in Fast Fourier Transform (FFT) hardware implementations, namely to redraw the trellis as sets of butterflies [Lei92]. The idea is shown in

5. Built-In Self-Test

5.1 Introduction to Built-In Self-Test

Since built-in self-test (BIST) [Bar87] is a promising method for testing large and complex integrated circuits without the need for very expensive testing equipment, BIST was incorporated into the design. In BIST, test pattern generation as well as output data evaluation are performed on the same chip as the circuit under test (CUT) [Bar87], [Abr90]. Figure 5.1 shows the general structure of BIST. A test pattern generator (TPG) can be implemented as a simple linear feedback shift register (LFSR) [Abr90], configured to output a maximum length pseudo-random sequence. To reduce the volume of the output data to be

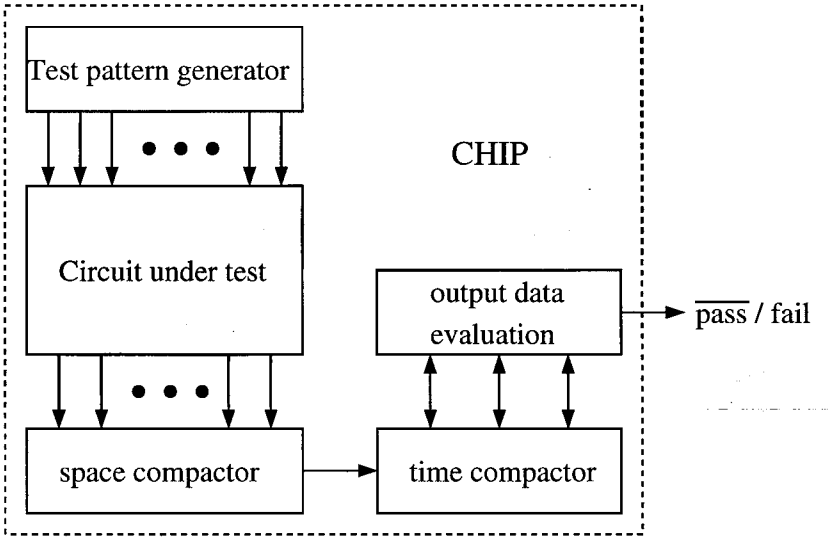


Figure 5.1 Block diagram of BIST.

evaluated, the output data of a multi-output CUT are compacted by a space compactor to fewer outputs. Some examples of generic space compactors are multiple input shift registers (MISR) [Abr90], multiple input no-feedback shift registers (MINSR) [Aga87], or XOR trees

[Kat92], [Li87]. More circuit specific space compactors are *programmable space compactors* (PSC) [Iva93], [Tsu93]. The customized nature of PSCs enables them to be more efficient in terms of hardware cost and lower error escape than generic space compactors. The space compacted data are fed into a time compactor to generate a final signature at the end of the test [Bar87]. The most common time compactors are LFSRs or MISRs with the contents of the shift register used as the signature [Bar87]. The signature is then compared with a fault-free reference to determine whether the circuit is good. Due to the information loss during data compaction, the problem of error escape or aliasing [Bar87] arises, where the signature generated by a faulty CUT is identical to the fault-free signature, thus causing a faulty circuit to be mistakenly declared good.

The fault model used in the BIST for this thesis is a simple *single stuck-at fault* model [Eld59] for gate inputs and outputs. Stuck-at fault means that due to a physical defect in the Si crystal or the fabrication process a node logically behaves as if it is connected to either V_{DD}, a stuck-at-1 (s-a-1) fault, or *GND*, a stuck-at-0 (s-a-0) fault. "Single" expresses the fact that we only deal with one faulty node at a time. Note the difference between fault and *error* [Abr90]. An error is the logical consequence of a fault at the primary outputs. For example, the stuck-at-0 fault at the input of an inverter causes the error of the output being "1" at all times. A fault is said to be *undetected* if it does not produce an error during the test period. The most common reasons for undetected faults are an insufficient set of test patterns or redundancy in the circuit [Bar87].

5.2 Multiple Signature Analysis

There exist two basic techniques for signature analysis [Bar87]: single signature (SS) analysis and multiple signature (MS) analysis. In the SS technique only one signature is compared at the end of a test session. Let l be the output sequence length and k the number of stages of the LFSR, and assuming equally likely errors, then the probability of aliasing

P_{al} is [Bar87]:

$$P_{al} = \frac{2^{l-k} - 1}{2^l - 1}. \tag{5.1}$$

For $l \gg k$, $P_{al} \approx 2^{-k}$, which is also true for unequally likely errors [Iva92]. The asymptotic result also holds true for MISRs if implemented with an irreducible polynomial [Kam93].

In the MS scheme, however, not only the final signature is checked, but also some intermediate signatures. This may greatly reduce the probability of aliasing. Let l_1, l_2, .., l_n for $k \ll l_1 < l_2 < ... < l_n = l$ be the positions where the signatures are checked, called *check points,* then the aliasing probability becomes [Bar87]:

$$P_{al}(n) \approx 2^{-nk}. \tag{5.2}$$

Moreover, as faulty chips can be discarded as soon as one incorrect signature has been detected test time in lower yield processes can be reduced significantly. A big disadvantage of the MS scheme, however, is its increased hardware overhead as checking n k-bit signatures requires n k-bit references to be stored. This implies a read-only-memory (ROM) for the references, address generation for the ROM, and of course scheduling for the right reference at the right time [Wu93a]. Solutions to reduce the hardware complexity in multiple signature analysis schemes are presented by Wu in his Ph.D. thesis [Wu93a] and his publications, e. g., [Wu92], [Wu93].

5.2.1 Fuzzy Multiple Signature Analysis

In [Wu92], Wu presented a fuzzy multiple signature (FMS) analysis scheme that greatly reduces complexity of a conventional MS scheme. Instead of comparing the intermediate signatures to their own references on a one-to-one basis, a signature only has to match any of the stored references. This fuzziness introduced by allowing more references to match an intermediate signature may result in a small increase of aliasing compared to the conventional

multiple signature scheme. However, this drawback can be compensated for by significantly less hardware complexity.

As an example, assume the checking of three 3-bit signature with the three references: r_1 = 010, r_2 = 111, and r_3 = 011. Denoting the three bits of the reference by *a, b, c*, we can describe the decision function as

$$\overline{pass} \,/\, fail = \overline{\bar{a}b\bar{c} + abc + \bar{a}bc} = \overline{bc + \bar{a}b} = \overline{\overline{ba\bar{c}}}, \tag{5.3}$$

which can be implemented with only one inverter and two NAND gates as shown in Fig. 5.2. Not only is no additional control logic required, but also can the $\overline{pass} \,/\, fail$ function

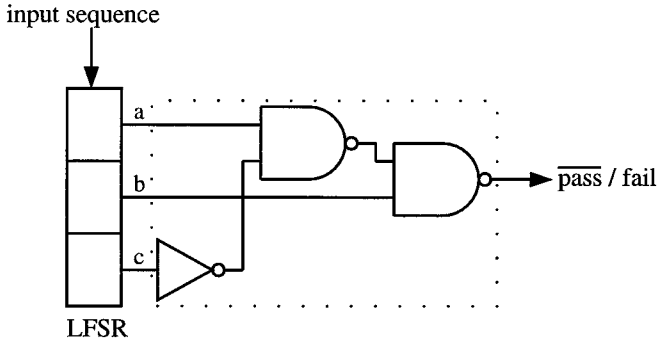


Figure 5.2 Example of the FMS scheme with the three references: 010, 111, 011.

some times be minimized [Wu92]. Moreover, some references may happen to be identical which results in a smaller number of references *m* than signatures *n* checked, i. e., $m < n$. To find the aliasing performance, it is necessary to consider the number of *l*-bit sequences that map to any one of the *m* references, namely $m2^{l-k}$, resulting in a probability of aliasing at a single check point [Wu92], [Wu93a]

$$P_{al} = \frac{m\,2^{l-k} - 1}{2^l - 1}. \tag{5.4}$$

Assuming $l \gg k$ yields $P_{al} \approx m2^{-k}$. The asymptotic result applied to *n* check points yields the aliasing probability for the fuzzy multiple signature scheme [Wu92]

$$P_{FMS} \approx \left(m2^{-k}\right)^n. \tag{5.5}$$

Clearly, the worst case occurs for $m = n$ (when all references are different). Then $P_{FMS} \approx \left(n2^{-k}\right)^n$. For $m = 1$, i. e., the case where all signatures are identical, the aliasing probability is equal to that of the conventional MS scheme. The following section is devoted to this identical signature case.

5.2.2 Minimal Hardware Multiple Signature Analysis

In the case of deliberately making all n signatures identical [Wu93] [Wu93a], only one reference has to be stored, and by spacing the n signatures equally apart ($l_1 = l_2 - l_1 = ... = l_n - l_{n-1}$), the hardware overhead for the timing controller is the same as for checking only one final signature [Wu93a]. The probability that a sequence generates a specific signature, say 0...0, is 2^{-k} and the probability that it generates n identical 0...0 signatures is 2^{-nk}. If we allow the signatures to have any value as long as they are identical, then the probability becomes 2^{-nk+k}. The problem however is that for practical values of k and n, the probability that all n signatures have the same value is very small.

If there is, however, a choice among many different sequences, then the probability of getting n identical signatures can be very high. Different sequences are achieved by applying different sets of test patterns to the CUT. Wu [Wu93] established a measure of confidence $C_{L,n,k}$ of finding n identical k-bit signatures, given L possible fault-free sequences, which follows a geometric distribution:

$$C_{L,n,k} = \sum_{i=1}^{L} \left(1 - 2^{-nk+k}\right)^{i-1} 2^{-n+1}, \tag{5.6}$$

or

$$C_{L,n,k} = 1 - \left(1 - 2^{-nk+k}\right)^{L}. \tag{5.7}$$

If L is made sufficiently large, n identical signatures can become highly probable. Solving Eqn. 5.7 for given n, k and a desired confidence C, gives the required L. Some results are

shown in Tab. 5.1.

L	C (%)
2^{nk-k+1}	86.47
2^{nk-k+2}	98.17
2^{nk-k+3}	99.97

Table 5.1 Desired confidence C versus required number of sequences L for a given $P_{al} = 2^{-k}$

A simple way to generate the required L different fault-free sequences is to apply L different sets of test patterns, for example, by starting the TPG with L different seeds. This approach requires simulating the CUT L times for l input patterns, where the simulation time complexity is of order $O(lL)$. A shortcut, however, is to shift the TPG L - 1 more times than what one test length l would be. This is equivalent to seeding the TPG L times and simulating the CUT with $l + L$ - 1 patterns, resulting in $l + L$ - 1 output bits [Wu93a]. In the search for identical signatures it is now possible to choose from L sequences of length l, one starting at the first output bit, another at the second, and finally, the last starting at the L^{th} output bit. The simulation time complexity for generating L sequences of length l is reduced to $O(l + L)$.

As an example, assume to check two identical signatures with the 3-stage LFSR shown in Fig. 5.3. Let l = 8, L = 3, and the $(l + L - 1)$-bit sequence to compact be:

$$1\,1\,0\,0\,1\,1\,0\,1\,0\,0. \tag{8}$$

Then the 10 intermediate LFSR states are:

$$(100)\,(110)\,(111)\,(011)\,(101)\,(010)\,(101)\,(010)\,(101)\,(110). \tag{9}$$

Checking every four bits, we find two identical signature of value (101) if the compactor

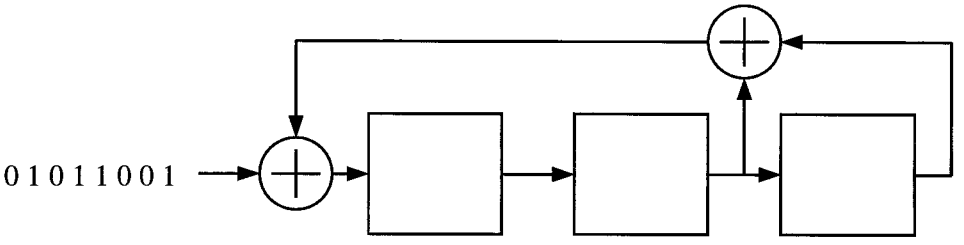


Figure 5.3 An example compactor.

LFSR is initialized with (100) and the subsequence 1 0 0 1 1 0 1 0 is compacted. The simulation also determines the initial state of the TPG to generate exactly the above subsequence.

Note that the number L of different sequences a TPG LFSR can generate is limited by its number of stages and possible internal states of the CUT if the CUT is not exclusively combinational. For example, a 5-stage LFSR can only generate a maximum of $L = 31$ different output sequences (the all-zero seed will not produce useful test patterns) of length $l = 31$. In general, it is not possible to generate more different output sequences than $2^k - 1$ because an LFSR can only be seeded with $2^k - 1$ different seeds, i. e., $L \leq l$. However, one can increase the probability of finding identical signatures by changing the seed in the compactor. Each stage of the compactor practically doubles L. Similarly, any flip-flop, independent from the TPG, such as free-running counters, can increase the effective number of sequences L. For practical larger circuits, which require many test patterns, the limitation $L < l$ is not a real concern.

The price that is to pay for better aliasing performance when using the minimal hardware overhead MS scheme, compared to the SS scheme, is the CPU-time overhead, including two parts. One is the time spent generating L fault-free sequences instead of only one. The other is the effort searching for the subsequence that yields n identical signatures. Compared to the effort of generating one l-bit fault-free sequence, the CPU-time overhead for generating

L sequences is proportional to L/l [Wu93]. However, consider that the CPU-time overhead is a one-time cost for recurring lower aliasing and area savings for each produced chip.

5.3 Implementation of the Minimal Hardware MS Scheme

In the Viterbi decoder described here, the minimal hardware MS scheme is applied to the BMU, the control block, the code memory, the majority gate, and the parallel-serial converter. In test mode, the feedback path in the control block is cut, and the control unit is reconfigured as a maximum-length LFSR to generate the test patterns. Fault simulations, using the SILOS II® fault simulator, showed that a 12-stage LFSR with feedback polynomial

$$x^{12} + x^7 + x^4 + x^3 + 1, \tag{5.10}$$

as shown in Fig. 5.4, generates enough pseudo-random test vectors for a complete test of the

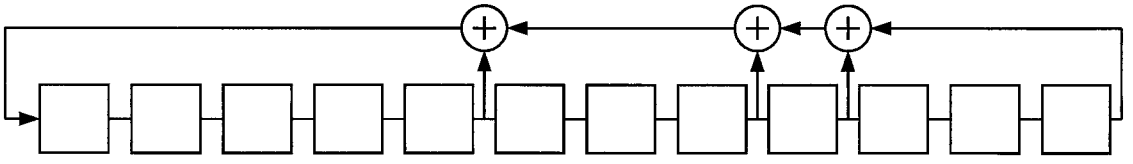


Figure 5.4 12-stage maximal-length shift register.

combinational part in the control block. The BMU and the code memory have less inputs than the combination part of the control block and hence are tested exhaustively. Interestingly, only 1024 test patterns, instead of all 4096, are needed to detect all single stuck-at-faults in the combinational part of the control block. The 12-stage LFSR is necessary to generate independent test vectors for the state outputs and their complements. Each of the sum terms in the combinational block, however, requires less than 12 independent inputs for exhaustive testing. The reconfiguration of the control block as a TPG is accomplished by multiplexers that cut the feedback lines and connect the flip-flops to form an LFSR in test mode. Figure 5.5a displays the control block in its mission mode, i. e., when TEST = 0. The multiplexers

M accept the state inputs and the flip-flops T output the state outputs to the combinational block. The double arrows symbolize busses. In test mode, when TEST = 1, the multiplexers

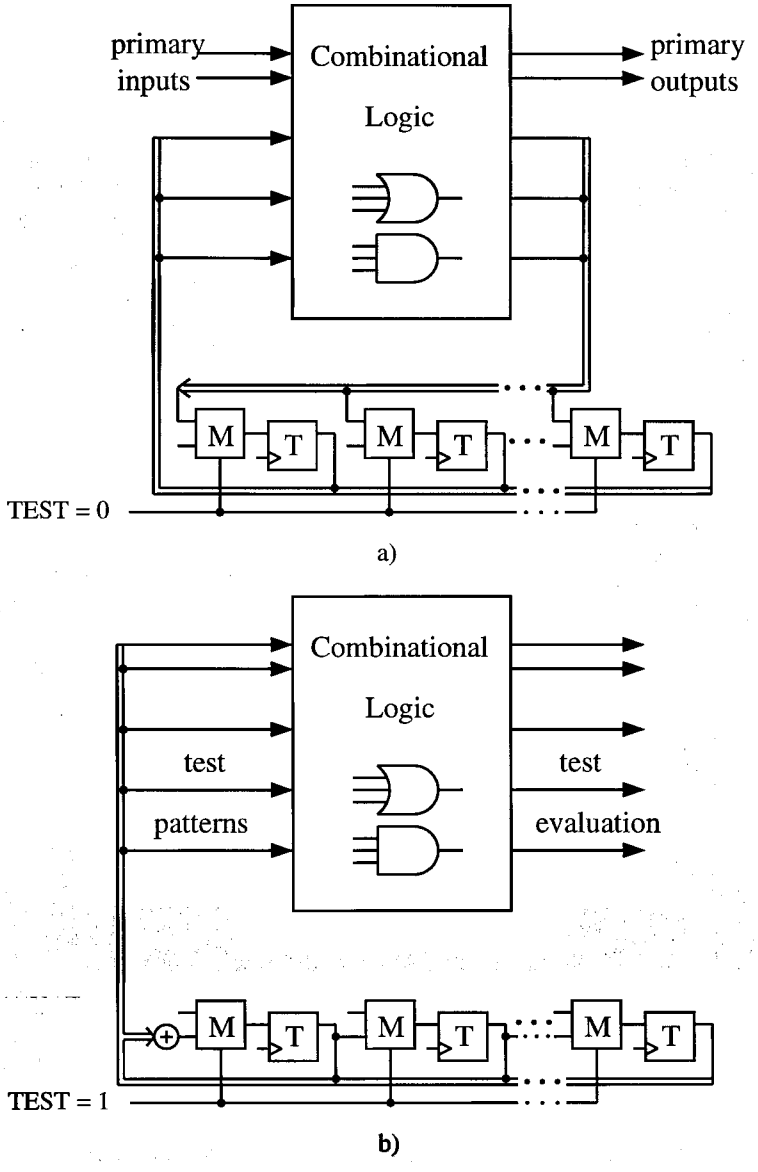


Figure 5.5 Reconfiguration of a) a finite state machine as b) a TPG LFSR in test mode.

accept the outputs from their predecessor flip-flops to form an LFSR. The $\oplus$ symbolizes the feedback lines for a maximum length pseudo-random output sequence.

Fault simulation determined where additional multiplexers were necessary in the BMU, the code memory, the parallel-serial converter, and the majority vote circuit to inject the random patterns in test mode for high fault coverage. Fault coverage is the percentage of faults detected in percent of the total number of faults considered. The achieved fault coverage is 98.2% of single stuck-at faults, i. e. 98.2% of the total of 8744 single stuck-at faults were detected. The reported fault coverage of 98.2% is before compaction in the MISR, because the SILOS® fault simulator cannot calculate the fault coverage after compaction. The main reason for not achieving a higher fault coverage is that the multiplexers that inject the test patterns are not tested for faults that appear in normal mode. The area overhead of the test scheme is about 10% of the circuitry tested, which seems relatively high. There are two reasons for that. First, the overall amount of circuitry tested with the MS scheme is relatively small, about a third of the chip. Therefore, the additional hardware for reconfiguring the test pattern generator and compaction counts heavily, percentage wise. The other reason is that some faults were hard to detect and needed extra multiplexers to test.

Recalling Table 5.1, the success of the minimal hardware scheme depends on the number L of generated sequences of length l to find one sequence with n identical k-bit signatures. The SILOS II® logic simulator could handle just over 100,000 or $\sim 2^{17}$ input vectors. With 86% confidence, one can find time compactors that have $nk - k = 16$. A good compromise between solutions with very low aliasing probability but considerable amount of hardware (e. g., two 16-bit signatures) and solutions with less hardware, but higher aliasing probability (e. g., 16 1-bit signatures) is to check four identical 5-bit signatures. This solution has a probability of aliasing $P_{al} = 2^{-20}$. As $nk - k = 15$, the chances of finding a solution rises to over 98%. An estimation of the fault coverage after compaction uses the fact that the expected number of escaped faults tends to $N2^{-k}$ [Raj91], where N is the total number of faults. With $N = 8744$ and $P_{al} = 2^{-20}$ used here, the number of aliased faults is $\ll 1$, which should not influence the overall fault coverage. It is desirable to check $n = 2^i$ signatures,

where i is an integer. In that case, the test length counter can be used to schedule the signatures very easily [Wu93a].

An XOR tree compacts 25 observation points into five bits, which are input into the 5-stage MISR with the characteristic polynomial of

$$x^5 + x^2 + 1. \tag{5.11}$$

SILOS II® simulates the circuit for all 100,000 clock cycles and provides the contents of the MISR. A small C program searches for four identical signatures, one every 256 clock cycles (Fig. 5.6) and provides the corresponding initial state of the test pattern generator and other

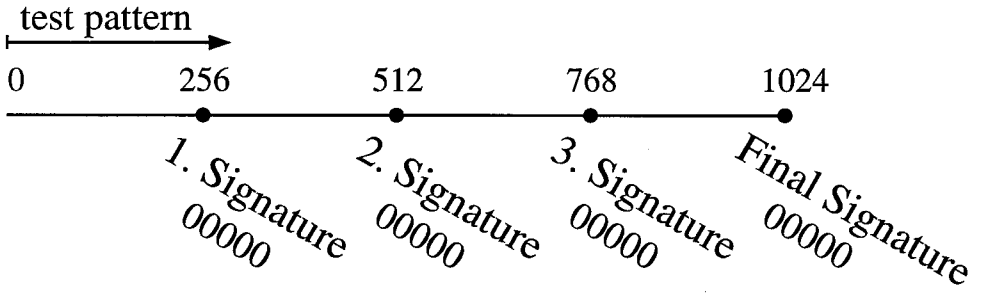


Figure 5.6 Four identical 5-bit (00000) signatures evenly spread in time.

flip-flops in the circuit. The signatures turned out to be the "all zero" state with an initial MISR state of 11010. Of course the signature can have any value. Simple circuitry resets all flip-flops to "0" in normal mode, but sets the appropriate flip-flops to "1" in test mode to the desired initial state. When a signature does not match the reference, a flip-flop sets the test output BAD to "1". If all signatures match the reference, BAD stays "0" and ENDTEST = "1" indicates the end of the test. The maximum clock frequency in test mode is reduced to 80 MHz, due to the extra delay in the XOR tree.

Compared to checking only one signature at the end of the test session, checking four identical signatures has the advantage of reducing the probability of aliasing here from 2^{-5} to 2^{-20}, while avoiding complicated signature checking for four different references. Lowering

the probability of aliasing also reduces the probability that unmodelled faults in the circuit be masked. Moreover, test time can be reduced as faulty chips can be discarded as soon as a signature does not match the reference. These advantageous features (reduced test time, low aliasing) are accomplished without recurring circuit overhead. The only cost is a one-time, logic simulation performed at design phase, about 45 CPU minutes on a SPARC 2 workstation to generate the L sequences and a few seconds to find the sequence yielding identical signatures.

5.4 Novel Test Scheme for the Register Exchange Path Memory

A short estimation of a pseudo-random test applied to all primary path-memory inputs shows that the test time for high fault coverage will be very long. The following analysis is a best-case estimate to demonstrate the poor observability of faults in a register-exchange path memory. Since a fault can propagate along many paths through the memory, it is easier to find an estimate of the probability that a fault will not propagate to the outputs compared to the probability that a fault will propagate to the outputs. Consider fault "One" in Fig. 5.7. The probability that the fault will not propagate through the memory element is P_{ni1} = 1/2, depending on the pseudo-random value at the multiplexer select line. The probability P_{ni2} that fault "Two" from Fig. 5.7 will not propagate through exactly two stages of the trellis is P_{ni2} = 1/2·1/4 = 1/8, since after it passed the first stage, the fault can propagate to one next state with probability 1/2, to both next states with probability 1/4, or not at all propagate with probability 1/4. Fault "Three" does not propagate through exactly three stages with probability

$$P_{ni3} = \frac{1}{2 \cdot 2 \cdot 4} + \frac{1}{2 \cdot 4 \cdot 16}. \tag{5.12}$$

The first addend accounts for the case where the fault propagates singly through two stages and is blocked at the third stage. The second addend covers the case the fault propagates

through the first stage, propagates to two states in the second stage, and does not propagate any further. The total probability that a fault does not propagate through three stages is the sum of P_{ni1}, P_{ni2}, and P_{ni3}. For the best-case estimation, only the cases where faults propagate to one next state will be considered here. Then, the probability P_{ni} that a fault does not propagate to the outputs of the path memory is

$$P_{ni} = \sum_{n=1}^{N+1} \frac{1}{2^n} - \frac{1}{4}, \tag{5.13}$$

where N is the number of stages a fault has to pass to reach the output. P_{ni} reaches 3/4 exponentially. That means that a fault will propagate to the output with probability of 1/4. Each path memory cell has 6 possible input stuck-at faults (the multiplexer has three inputs), and there are 1152 memory cells in our path memory. To propagate the total of 6912 input stuck-at faults on average four times as many (=27,648) test vectors will be necessary.

A similar estimation can be done for output stuck-at faults, such as fault "Four" in Fig. 5.7. Here, the probability P_{no} that the fault does not propagate to the output reaches 1/2 exponentially, again only considering paths where faults propagate to one successive state

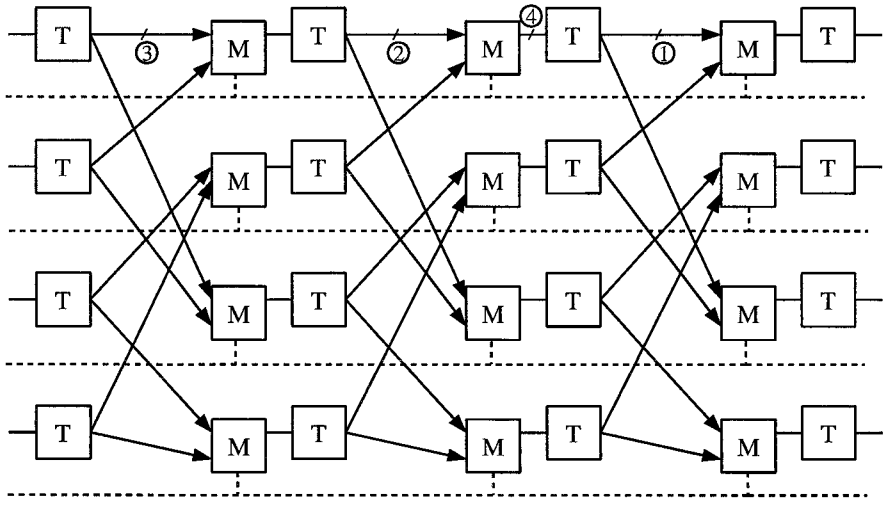


Figure 5.7 Four example faults.

before being blocked. In the register-exchange path memory faults of three nodes fall into that category (output multiplexer, input flip-flop, output flip-flop), yielding a total number of stuck-at faults of 6912. Here, on average twice as many (=13824) test vectors as faults need to be applied to propagate these faults.

To propagate all faults to the output a total of 27,648 + 13824 = 41,472 test vectors will be necessary. This number is certainly a lower bound, because:

- A fault can go through many more possible paths before it does not propagate further;
- A 16-stage LFSR to generate the pseudo-random inputs will not even generate many of the required pattern combinations to propagate a fault to the outputs;
- It is still not guaranteed that all memory cells receive the necessary inputs to test the stuck-at faults.

Compared to the 1024 test vectors necessary for the multiple signature analysis scheme described in Section 7.3, this number is very big and would increase test time by at least one order of magnitude. However, the regular structure of the path memory suggests a deterministic test to decrease test time. The following test patterns are sufficient to completely test a memory element and a multiplexer.

— D-flip-flop: A memory element is completely tested by "load 0", "hold 0", "load 1", "hold 1", and "load 0" [Bar87]. Since the memory cells here are flip-flops, the flip-flops also have to be tested for s-a-0 faults and s-a-1 faults at their clock inputs.

— Multiplexer: A multiplexer is characterized by its inputs A and B, its select line SB, and its output OUT. If SB = 0, A is selected, if SB = 1, B is selected. Table 5.2 displays the four necessary input pattern combinations to test for all eight

possible single stuck-at-faults in a multiplexer. Faults in internal nodes of

Input pattern (A B SB)	detects
0 1 0	A-stuck-1, OUT-stuck-1, SB-stuck-1
0 1 1	B-stuck-0, OUT-stuck-0, SB-stuck-0
1 0 0	A-stuck-0, OUT-stuck-0, SB-stuck-1
1 0 1	B-stuck-1, OUT-stuck-1, SB-stuck-0

Table 5.2 Four input patterns detect all possible eight stuck-at-faults of a multiplexer.

the multiplexer create contentions at the output node, as both transmission gates have one transistor "ON" and one "OFF". The output may have any value and therefore the fault may or may not be detected.

5.4.1 Test Algorithm for Path Memory

When developing a test scheme for such a large block as the path memory, it is desirable to introduce as much parallelism as possible. The structure of the trellis suggests applying complement values to the multiplexer select lines of multiplexers in states with different MSBs, as can be seen in Fig. 5.8. Then, the half of the multiplexers with select lines "0" selects the upper branches entering a node, and the other half with select lines "1" selects the lower branches entering a node. Together, the multiplexers select one leaving branch from each node from the previous memory stage. Hence, the value of each node will propagate to the next trellis stage. The "0"s and "1"s at each node in Fig. 5.8 indicate the select values for the multiplexers. The solid lines show which of the trellis branches are selected.

In the following, it is not important which fault is tested at what time, as long as every multiplexer receives all four different input test patterns from Tab. 5.2 during the test session. Thus, at any time during the test, the multiplexer inputs A and B should be complements. Let s_0, s_1, ..., s_{N-1}, with $N = 2^{K-1}$, be the input values to each of the N states. The subscripts denote the numbers of the states. Then, one out of two possible input patterns,

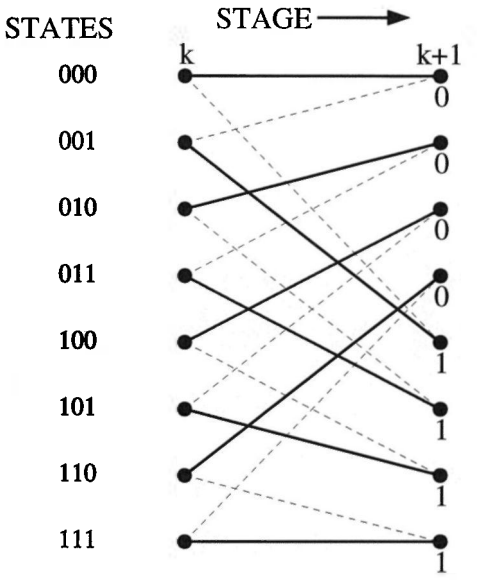


Figure 5.8 Applying complement values to the multiplexers of different halves propagates the values of each state to the next stage.

denoted by *input pattern*, is obtained by the following procedure *find.input.pattern*, written in pseudo code.

```
find.input.pattern
    s0 = 0; i = 0; j = 0; k = 0;
    while(2^i < N) do
        for(j = 2^i to 2^(i+1) - 1) do
            sj = s̄k;
            k = k + 1;
        endoffor
        i = i + 1;
        k = 0;
    endofwhile
endoffind.input.pattern.
```

The other possible input pattern is simply obtained by complementing the pattern obtained by the procedure *find.input.pattern* or, by starting the procedure with $s_0 = 1$ and is denoted by

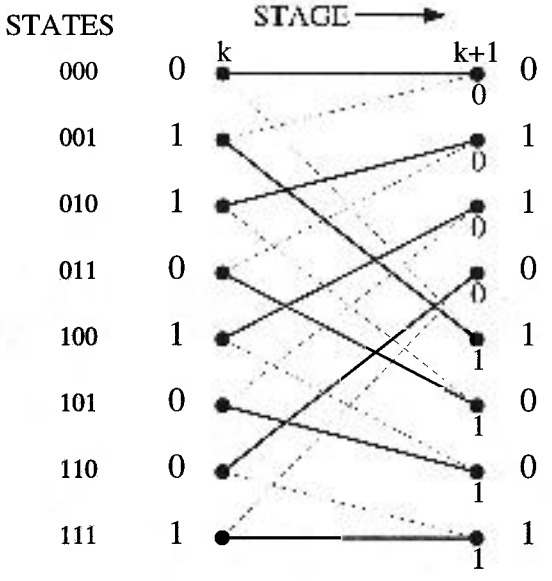


Figure 5.9 Example of an input pattern for the path memory test.

$\overline{input\ pattern}$. Figure 5.9 shows as an example $input\ pattern$ for an eight-state trellis. This input pattern ensures that the same pattern is transferred to the next stage, if the multiplexer select lines are "0" for states with MSBs = "0" and "1" for states with MSBs = "1", denoted $select\ pattern$. Let the complement pattern, i. e., when the multiplexer select lines are "1" for states with MSBs = "0" and "0" for states with MSBs = "1", be $\overline{select\ pattern}$.

The procedure *test.pathmemory* contains the pseudo code for the complete test algorithm for the register exchange path memory. The path memory is of length d.

test.pathmemory(*input pattern, select pattern, d*)
for(i = 1 **to** *d*) **do**
apply *input pattern* and *select pattern*
endoffor
apply $\overline{input\ pattern}$ and $\overline{select\ pattern}$
for(i = 1 **to** (*d* - 1)) **do**
apply $\overline{input\ pattern}$ and *select pattern*
endoffor
apply *input pattern* and $\overline{select\ pattern}$
for(i = 1 **to** (*d* - 1)) **do**
apply *input pattern* and *select pattern*
endoffor
endoftest.pathmemory.

Four snapshots of the test algorithm are shown in Fig. 5.10. The first "for" loop of the procedure *test.pathmemory* loads the *input pattern* into the path memory, as can be seen in Fig. 5.10a. The multiplexers are tested for one test pattern from Table 5.2. For example, the multiplexer of state 000 receives the test pattern 010, the multiplexer of state 001 is tested with the pattern 100. When applying $\overline{input\ pattern}$ and $\overline{select\ pattern}$ once, as shown in Fig. 5.10b, all multiplexers are tested for a second test pattern in parallel. The multiplexer of state 000 receives the test pattern 011, the multiplexer of state 001 is tested with the pattern 101. Note that though $\overline{input\ pattern}$ is applied to the input of the path memory, the multiplexers inside the path memory receive the *input pattern*. The application of $\overline{input\ pattern}$ and *select pattern* $d - 1$ times tests the multiplexers with a third test pattern (Fig. 5.10c) and propagates possible errors to the output and test the multiplexers for a third test pattern. The fourth and last test pattern for the multiplexers, as shown in Fig. 5.10d (101 for state 000 and 011 for state 001) is generated by applying *input pattern* and

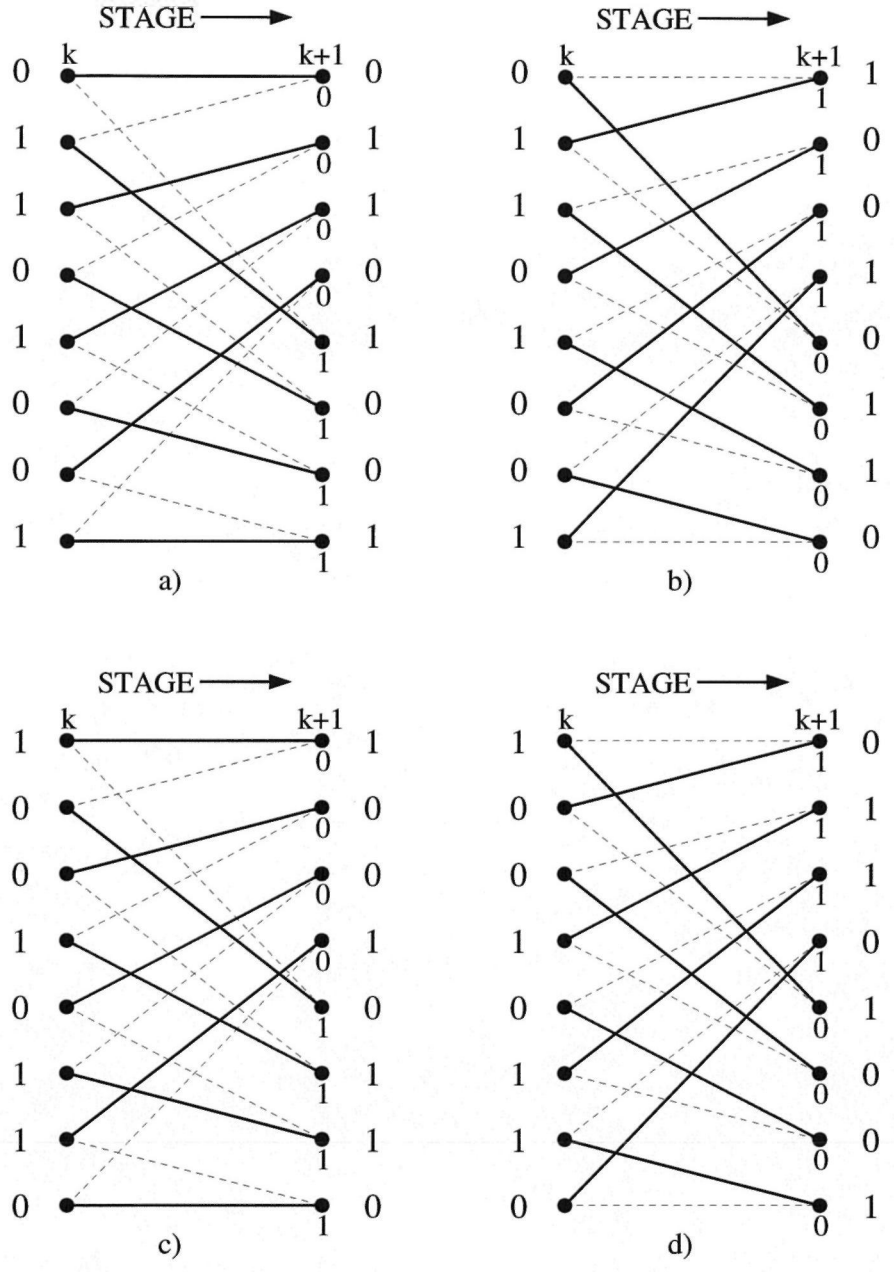


Figure 5.10 Four sets of test patterns are input to the path memory.

$\overline{select\ pattern}$ once. To shift out any detected error in the path memory, *input pattern* and *select pattern* are applied $d - 1$ times.

During the test procedure *test.pathmemory*, not only were the multiplexers tested with all necessary test patterns from Table 5.2, but also the flip-flops were tested for loading and holding "0"s and "1"s. Stuck-at-1 faults at the clock inputs are detected since the faults create races. For stuck-at-0 faults, the outputs of the flip-flops are floating and do not produce useful values, hence s-a-0 faults are detected.

From the pseudo code in procedure *test.pathmemory* it becomes apparent that the total test length is three times the path memory length and is independent of the number of states of the Viterbi decoder. A major advantage of this test algorithm is its simple set of input vectors; only two different patterns for the multiplexer inputs and two patterns for the multiplexer select lines. A very efficient implementation with slightly increased test time uses the test counter to generate the test patterns. The i^{th} stage, where $2^i > 2d$, generates a constant output at least d times. Inverters to the inputs of the path memory create the *input pattern* according to the procedure *find.input.pattern*. A decoder in the counter outputs a single "1" to invert the $select\ pattern$ once, when stage i has a transition.

5.4.2 Output Data Evaluation

Section 5.4.1 described a very efficient test algorithm for all possible single stuck-at faults in a register exchange path memory. The fact that adjacent outputs of the path memory are always complements can be exploited in a bit-by-bit comparison of the outputs. Then, an XOR tree always outputs a "0" in the fault free case during a test session. This can be used to compare the XOR tree output to "0" every clock cycle and eliminate any aliasing. The XOR tree does not change the fact that the algorithm can detect any single stuck-at fault because the XOR tree propagates single errors. In fact, the scheme will detect any odd number of faults as long as the faults happen to be only in the path memory and not also in the XOR tree. In the case of any even number of faults they will only remain undetected if they produce the same error at the same stage (or pairs of stages) of the path memory,

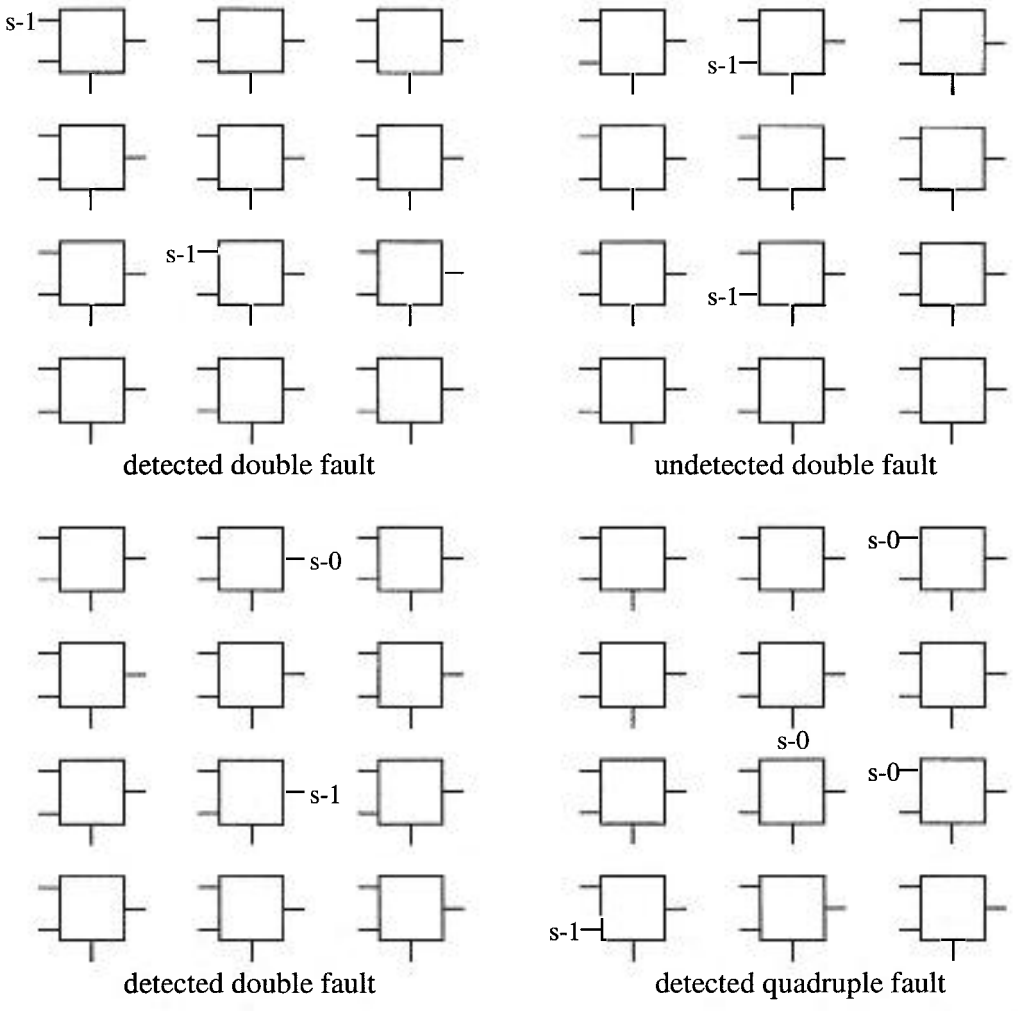


Figure 5.11 Examples of detected even number of faults and an undetected double fault.

so that they cancel out in the XOR tree. Errors in different stages appear at the output at different times and will therefore be detected by the XOR tree. Figure 5.11 shows some examples of double and quadruple faults that are being detected and one that escapes. The faults are abbreviated with s-1 for stuck-at-1 and s-0 for stuck-at-0.

A critical situation may happen if say for example the output of the XOR tree is stuck 0. Then, the test will always say "pass". This is no problem as long as this fault is the only fault in the path memory, since the XOR tree has no role in mission mode. Any additional

fault in the path memory, however, will also stay undetected and thus cause the circuit be mistakenly declared good.

A major improvement on the fault coverage for multiple stuck-at faults is possible by substituting the XOR tree after the first column of XORs with a big AND gate and compare to a "1". Since adjacent outputs of the path memory are always complements, a parity comparison of two adjacent outputs will always yield a "1". Any number of faults in the path memory itself are detected, because the AND gate outputs a "1", only if all inputs are "1". Again, however, if the output of the AND gate is stuck-at 1, no faults can be detected.

5.4.3 Algorithm Performance

In the case of an XOR tree as space compactor, the fault coverage for single stuck-at faults after compaction is the same as before compaction, because the XOR tree does not introduce any aliasing. A path memory storage element, consisting of a multiplexer and a D-flip-flop with two clocks can have 16 single stuck-at faults at the gate level. Then, the total number of single stuck-at faults in the path memory is 18,720, including the clock-distribution tree. The overall detection rate after compaction for single stuck-at faults is 99.74%. The reason for not being 100% is that the multiplexers that inject the test patterns are not tested for faults in normal mode.

The fault coverage for double fault can be easily calculated by first finding the total number of double faults and then the number of escaped double faults. Since one storage element has 8 pins (4 from multiplexer and 4 from D-flip-flop), the total number of nodes n in the path memory is n = 8 x 16 x 72 = 9216. The total number of double faults is $2n(n - 1)$ = 169,850,880. The number of double faults escaping is only the number of multiplexer select nodes times the number of nodes in the space compactor (21), beginning at the outputs of the XORs. The total number of undetected double faults is 24,192, yielding a fault coverage of 99.98%.

The area overhead of this high performance test scheme is only 2% of the original path memory area, including the multiplexers, the XOR tree, and extra wiring area.

5.5 Test for the Add-Compare-Select Block

The ACS block can be tested by a similar bit-by-bit comparison as the path memory. Fault simulations showed that only two pseudo-random patterns have to be injected from the test pattern generator into each of the ACSUs instead of the path metrics that are fed back to achieve complete fault coverage of the ACSUs. The branch metrics (BM 00, BM 01, BM 10, BM 11), the symbol clock, and the ACCEPT PATH signal are pseudo-random in test mode and do not need extra multiplexers for test patterns. Identical pseudo-random patterns are injected into the ACSUs in parallel.

Figure 5.12 demonstrates the idea with an example of an eight-state Viterbi decoder with butterfly-paired ACSUs. The branch metric input labels are based on the encoder of Fig. 2.1. Each ACSU, that receives the same branch metrics in the same order, e. g., top: BM 00, bottom: BM 11, generates identical output sequences. Viterbi decoders of eight states or more have at least two ACSUs that generate the same output sequences. In Fig. 5.12 states 000 and 101, 100 and 001, 010 and 111, and finally 110 and 011 produce identical output sequences. In this example, the output sequences after the second stage of an XOR tree provide the all-zero sequences. The observation that the XOR tree outputs a zero, even if the BM connection, the symbol clock connection, and the ACCEPT PATH connection was not changed, saves as much as four multiplexers in each ACSU.

Any detected "1" at any time in the XOR-tree output indicates that an error has been detected, and the chip can be discarded. The advantage of using an XOR tree as a space compactor and no time compactor is that the XOR tree does not introduce aliasing for any odd number of faults. Even numbers of faults remain undetected only when they produce the same error at the same time as explained in Section 7.4.

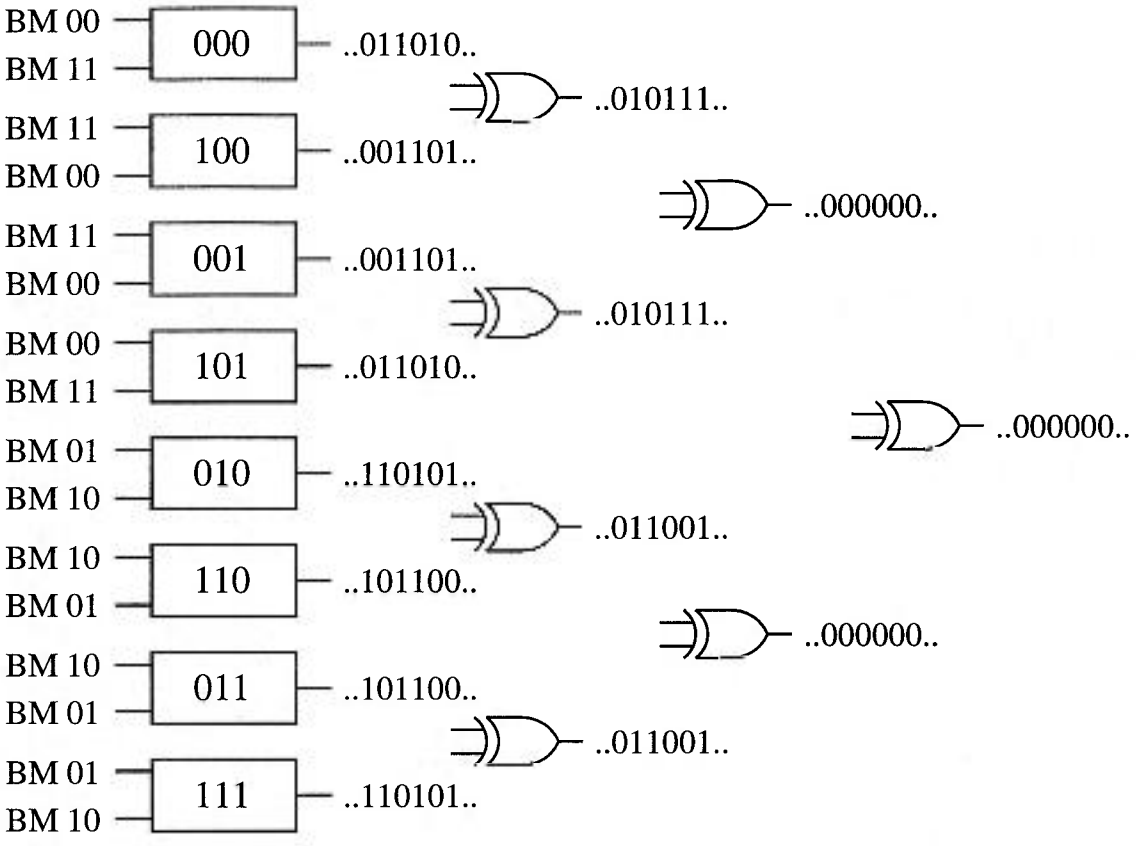


Figure 5.12 Example of the bit-by-bit comparison in the ACS block.

The total number of single stuck-at faults in the ACSU block including the XOR-tree-space compactor is 3502. The achieved fault coverage is 98.6% after compaction. The area overhead of the test is 5% of the ACSU block area, again including the XOR tree and wiring area.

6. Prototype Chip

6.1 Chip Specifications

The Viterbi decoder has been integrated on a single chip in 1.2μm CMOS double metal layer technology, partly using a standard cell library. The total chip size is 3.9 mm x 3.6 mm, or 14 mm^2. The active core area is 9.5 mm^2. The chip contains about 29,000 transistors of which 19,000 are in the custom layout blocks. The expected clock speed is 100 MHz to achieve a maximum decoding speed of 12.5 Mbits/s channel rate. For different code rates this translates into 3 Mbits/s to 11 Mbits/s information rate. The chip has 30 pins of which 6 are for power supply. The operation of the Viterbi decoder requires 11 pins, 3 more are needed for the built-in self-test. 10 extra pins are provided for debugging possible design errors.

6.2 Design Tools

The VLSI CAD system EDGE™ from CADENCE™ provided by the Canadian Microelectronics Corporation was used. Simulations of the standard cells were carried out with SILOS II® logic simulator and custom layouts were simulated in HSPICE™. The SILOS® fault simulator determined the fault coverage for the built-in self-test. Design, layout and simulations were carried out on SUN SPARC 2 and SUN SPARC 10 work stations provided by the Canadian Microelectronics Corporation.

6.3 Pin Description

The arrangement of the pads on the chip can be seen from Fig. 6.1. Arrows toward

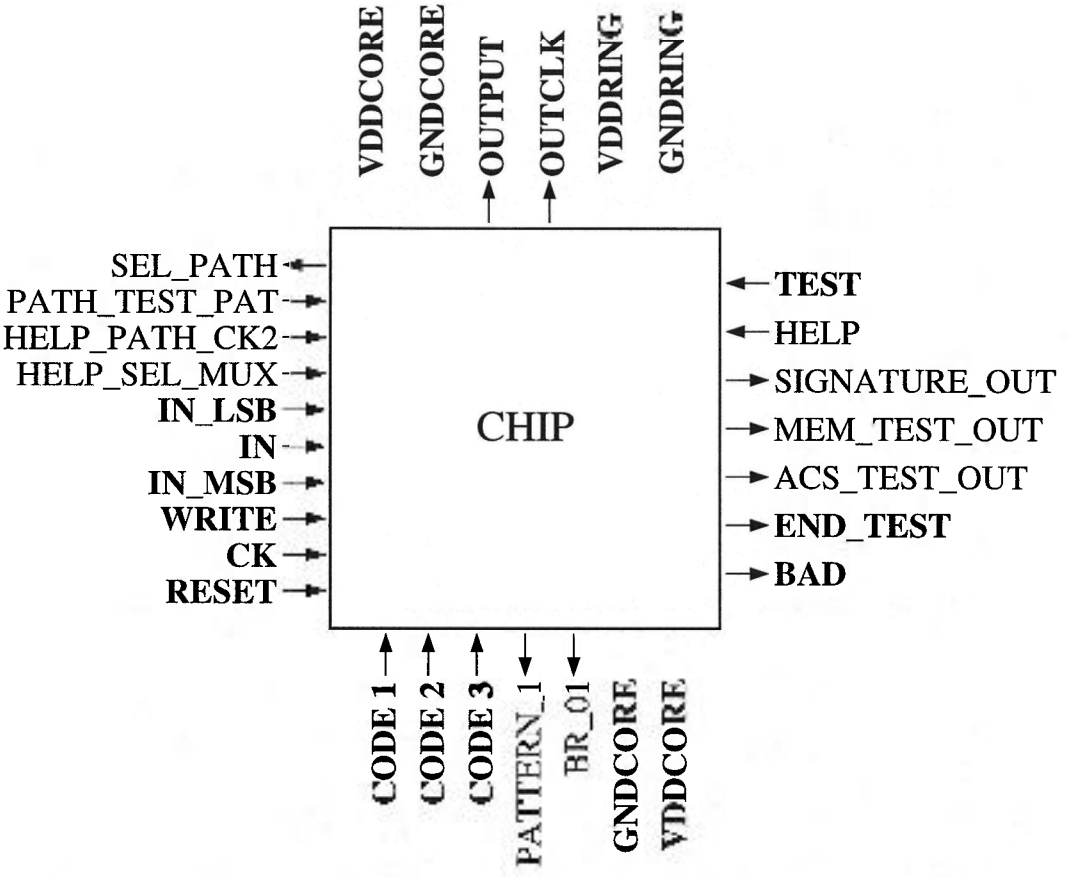
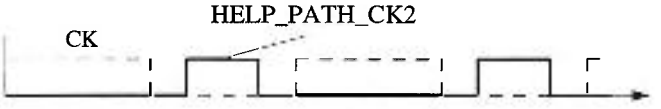


Figure 6.1 Pad placement on the chip.

the chip indicate inputs, arrows pointing from the chip denote outputs. A commercially produced chip would have only the bold face pads. They alone are necessary to form a fully testable Viterbi decoder for variable rates. The additional pads increase observability and controllability of the chip. In the case of a design error they help to narrow down the location of the error. The following is a list of all pins and the corresponding description. The number in the last column refers to the pin number of a 68-pin pin grid array as will be shown in Appendix B.

Pad Name	Description	Pin #
ACS_TEST_OUT	*Output.* Is the output of the bit-by-bit comparison of the ACS block test. ACS_TEST_OUT = 1 if an error was detected in the ACS block.	23
BAD	*Output.* Is the test result of the complete BIST. BAD = 1 if an error was detected in the chip.	19
BR_01	*Output.* Outputs the value of the branch metric BM 01.	10
CK	*Input.* Provides the overall chip clock. The required clock format is "10" for one clock cycle.	66
CODE 1, CODE 2, CODE 3	*Inputs.* Input the current code matrix here one column at a time. CODE 1 corresponds to the MSB of the code mapping (see Section 4.4). The code columns are entered in channel symbol speed. A maximum of seven columns are allowed.	2, 4, 6
END_TEST	*Output.* Indicates the end of the self test. The chip is said to be good if BAD = 0 and END_TEST = 1.	21
GND_CORE	*Ground.* Connects to the core area.	12, 46
GND_RING	*Ground.* Connects to the pad ring.	38
HELP	*Input.* Control point to allow a external test of the path memory, if HELP = 1.	29

HELP_PATH_CK2	*Input.* Provides the second non overlapping clock for external path memory testing if HELP = 1. Clock 1 is the chip clock CK. 	56
HELP_SEL_MUX	*Input.* Takes the values for the multiplexer select lines for external path memory test, when HELP = 1. HELP_SEL_MUX is directly connected to the upper half and via an inverter to the lower half of the multiplexer select lines.	58
IN_MSB, IN, IN_LSB	*Inputs.* The eight-level quantized channel symbols to be decoded.	62, 61, 60
MEM_TEST_OUT	*Output.* Outputs the result of the bit-by-bit comparison of the path memory test algorithm.	25
OUTCLK	*Output.* Provides the clock of the the decoded information bits. Can be used as write clock for a first-in-first-out memory.	42
OUTPUT	*Output.* Outputs the decoded information bits.	44
PATH_TEST_PAT	*Input.* Takes the input pattern for external path memory test, when HELP = 1. Connects to all path memory inputs.	54

PATTERN_1	*Output.* One output of the pseudo-random test pattern generator.	8
RESET	*Input.* Global reset. All flip-flops are reset to state zero in normal mode (when TEST = 0). In test mode (when TEST = 1) it sets the circuit into the initial state for identical signatures.	68
SEL_PATH	*Output.* Output of the control block that is "1" when the path metrics are selected at the inputs of the ACSUs and is "0" when the partial path metrics are recirculated.	52
SIGNATURE_OUT	*Output.* Gives the result of the multiple signature analysis.	27
TEST	*Input.* Switches between normal operating mode (TEST = 0) and test mode (TEST = 1).	31
VDD_CORE	*Power.* 5 V. Connects to the chip core.	14, 48
VDD_RING	*Power.* 5 V. Supplies the I/O pads with power.	40
WRITE	*Input.* WRITE = 1 enables the user to enter the code matrix. Each code column takes one channel symbol period (= 8 clock cycles).	64

6.4 Chip Layout

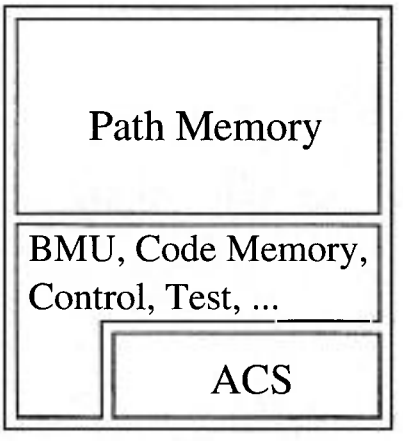


Figure 6.2 Floor plan of the Viterbi decoder.

Figure 6.2 shows the floor plan and Fig. 6.3 the metal 1 and metal 2 layers of the fabricated chip. The path memory occupies almost half of the chip area and is placed on the upper half of the chip. The ACS block can be seen in the lower right corner. The rest are standard cells, placed by the Cadence™ Place and Route routine.

The fabricated chip is mounted on a 68-pin pin grid array package (PGA). Appendix B shows the bonding diagram and the top and the bottom view of the packaged chip. The circled pins indicate used pins.

6.5 Test Results

To verify that the Viterbi decoder is working in theory, the circuit was simulated with SILOS II® using encoded data as simulation input patterns. For the ACS block and the path memory, which exist only as their layout, equivalent standard cell blocks were substituted. Repeating these simulations with several different code rates, raised the confidence of proper functionality.

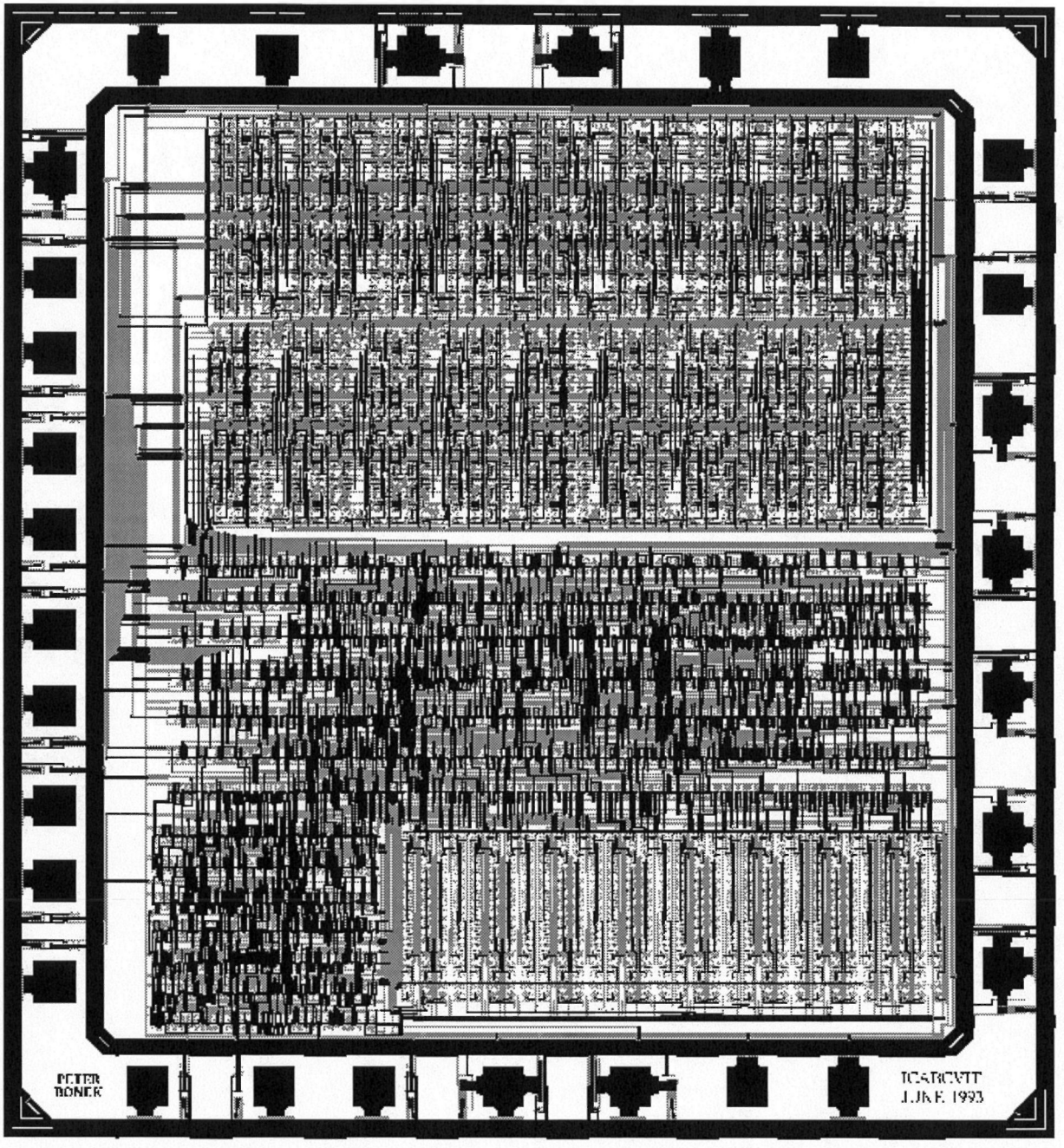


Figure 6.3 Metal 1 and metal 2 layer of the chip that show the path memory on the upper half and the ACS block in the lower right corner. The rest is placed by the Cadence™ Place and Route.

Unfortunately, the Place and Route function had a software error and did not connect the output pads of the chip properly. The output pads have to be enabled by connecting the ENABLE pin to V_{DD}. Place and Route connected all ENABLEs together, but left them floating. It is not possible to see any output signals of this prototype chip. Hence test results are not available.

However, even without working output pads, estimations of the power consumption are possible. The power consumption was determined by measuring the average current flowing into the chip and multiplying by the supply voltage of 5V. The measurement was conducted in two series. First, all four prototype chips were tested in normal mode for decoding an imaginary 7/8 rate code. The 7/8 rate code gives the upper limit in power consumption in mission mode, since 7 decisions are made in the ACSUs for 8 incoming channel symbols, resulting in the highest clock frequency for the path memory flip-flops. A counter generated the imaginary input symbols. Then, tests were repeated in test mode. In test mode, no inputs are necessary (except a static TEST = 1), since all test patterns are generated on-chip.

Since it is not possible to verify that the chip is working correctly, the power consumption was also estimated by HSPICE™ simulations. Since it was not possible to simulate the complete Viterbi decoder in one piece with HSPICE™, smaller blocks were simulated separately. The total power dissipation is the sum of the power consumptions of all blocks. Adding a simple "power meter", as can be seen in Fig. 6.4, eases the estimation of the power consumption. The current-controlled current source is controlled by the current flowing through node V_{DD} and loads a capacitor. The integrated voltage at the capacitor, multiplied with a scaling factor, is a measure of the used power in pWatts. The resistor is necessary to provide a DC path to ground for the HSPICE™ simulation.

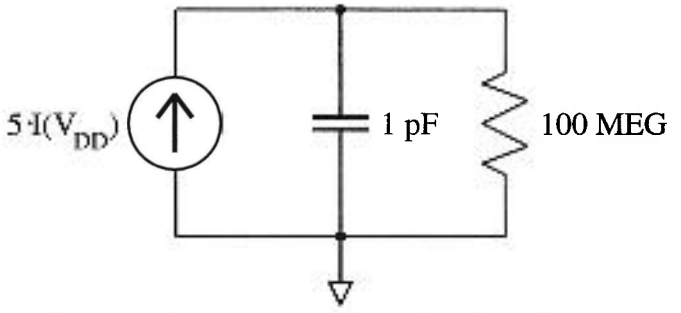


Figure 6.4 "Power meter" for HSPICE™ to estimate the power consumption.

Figure 6.5 shows the simulated, measured, and from the measurements extrapolated power consumption of the Viterbi decoder. The measured power dissipation of all chips was

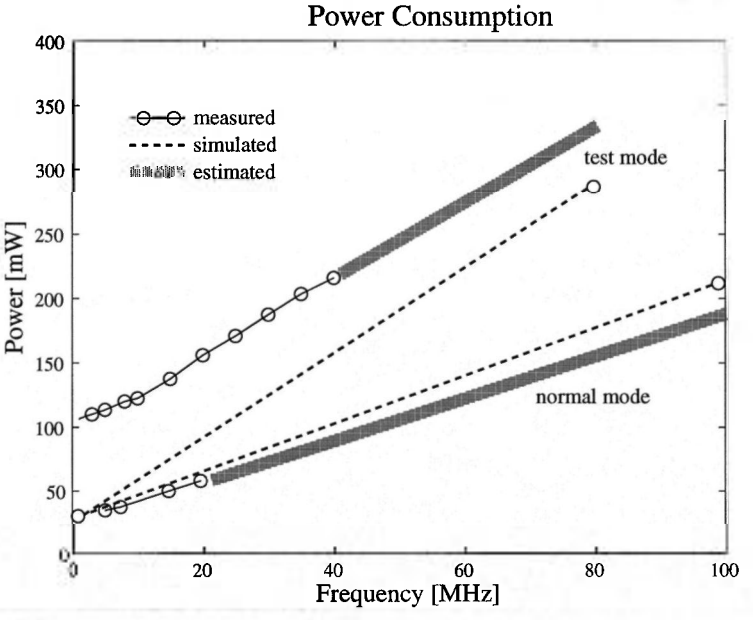


Figure 6.5 Estimated power consumption up to the maximum frequencies.

within +/- 2 mW. To leave the figure readable, only the mean is shown in Fig. 6.5. For normal mode both simulations and extrapolated measurements show good agreement over the entire frequency range. 200 mW at maximum speed is reasonable for battery operated mobile devices. The power consumption is not zero when the clock frequency is zero due to a static power consumption of 30 mW in the path memory. The major disagreement of measured

and simulated power dissipation in test mode cannot be explained without the actual output data to determine if, and if where, a mistake has happened that could explain the discrepancy.

7. Conclusion

This work described the development and design of a variable-rate Viterbi decoder of constraint length $K = 5$ (memory order $m = 4$). The decoder supports any code rate ranging from 7/8 to 1/4. The chip has been implemented in Northern Telecom's 1.2 μm CMOS double metal layer process and occupies a total of 14 mm^2. The architecture of the Viterbi decoder is bit-serial node-parallel. The incoming 3-bit quantized channel symbols are input in parallel, converted to a serial stream, and processed serially. This reduces the amount of interconnect area substantially, as the add-compare-select (ACS) units are connected by single wires only. High decoding speed is still achieved because the ACS operation is carried out concurrently in each of the 16 states.

The decoder uses a modulo normalization of the path metrics that can be performed within each ACS unit to prevent register overflow errors. This eliminates the need for additional circuitry and long global wires. Custom layout of the complete ACS block and butterfly pairing of ACS units reduced the silicon area to less than 50% of a standard cell implementation.

The path memory is implemented using the register exchange technique. This technique was adopted as it occupies less silicon area than a traceback method implementation. A traceback path memory, however, is very attractive, if silicon area is not the first concern. The advantage of a traceback memory, if its speed is sufficient, is that it can be implemented using standard cell design automation tools and a tested macro cell static RAM. The register exchange technique is based on the movement of information sequences through the path memory, accomplished by trellis connected shift registers, one for each state. To reduce the

silicon area, the path memory is full custom layout. For the trellis interconnections between consecutive memory stages, a new state relabelling technique that reduces the interconnect area to 30% compared to a straightforward trellis wiring was proposed and adopted for the design. The area reduction was accomplished by redrawing the trellis as sets of butterflies.

A major aspect of this chip is its very cost effective built-in self-test. The fault coverage of the complete test before compaction for single-stuck-at faults at the logic gate level is 99% with a hardware overhead of only 5%. A very efficient, novel test algorithm is used for the path memory. The test complexity is independent of the constraint length of the Viterbi decoder and the specific input pattern can be derived from the test counter very easily. Since adjacent outputs of the path memory always produce complement values in test mode, a bit-by-bit comparison that avoids any aliasing can be used for output data evaluation. The ACS block is tested by a similar bit-by-bit comparison. However, the input patterns are pseudo random. The rest of the circuit is tested by pseudo-random patterns and a multiple signature analysis scheme. After finding an appropriate initial state of the test pattern generator and other flip-flops in the circuit, it is possible to check four identical signatures. Compared to checking only one signature at the end of the test session, checking four identical signatures has the advantage of reducing the probability of aliasing, while avoiding complicated signature checking for four different references. Moreover, test time can be reduced as faulty chips can be discarded as soon as a signature does not match the reference. The only cost is a one-time logic simulation performed at design phase, while saving recurring circuit overhead.

References

[Abr90] M. Abramovici, M. A. Breuer, A. D. Friedman, *Digital Testing and Testable Design*, Computer Science Press, New York, 1990.

[Aga87] V. K. Agarwal, Y. Zorian, "An Introduction to an Output Data Modification Scheme", in *Development in Integrated Circuit Testing*, Academic Press Limited, 1987, pp. 219-256.

[Bar87] P. H. Bardell, W. H. McAnney, J. Savir, *Built-In Test for VLSI: Pseudorandom Techniques*, New York: John Wiley & Sons, Inc., 1987.

[Bha81] V. K. Bhargava, D. Haccoun, R. Matyas, P. Nuspl, *Digital Communications by Satellite*, John Wiley & sons, New York, 1981, pp. 353–402.

[Bre92] M. A. Bree, D. E. Dodds, R. J. Bolton, S. Kumar, B. L. F. Daku, "A Modular Bit-Serial Architecture for Large-Constraint-Length Viterbi Decoding", *IEEE J. Solid-State Circuits*, vol. SC-27, pp. 184-190, February 1992.

[Cai79] J. B. Cain, G. C. Clark, Jr., J. M. Geist, "Punctured Convolutional Codes of Rate *(n-1)/n* and Simplified Likelihood Decoding", *IEEE Trans. Inform. Theory*, vol. IT-25, pp. 97-100, January 1979.

[Cha89] H. J. Chao, C. A. Johnston, "Behavior Analysis of CMOS *D* Flip-Flops", *IEEE J. Solid-State Circuits*, vol. SC-24, pp. 1454-1458, October 1989.

[Cog88] D. J. Coggins, B. S. Vucetic, D. J. Skellern, "Trellis Partitioning Scheme Based on State Relabelling for Viterbi Decoding in VLSI", in *International Symposium on Information Theory*, IEEE Information Theory Group, p. 228, 1988.

[Cog89] D. J. Coggins, D. J. Skellern, R. A. Keaney, J. J. Nicolas, "A Comparison of Path Memory Techniques for VLSI Viterbi Decoders", *VLSI'89, Proc. of the IFIP TC*

10/WG 10.5 Int. Conf. on Very Large Scale Integration, pp. 379-388, Munich, FRG, August 1989.

[Col92] O. M. Collins, "The Subtleties and Intricacies of Building a Constraint Length 15 Convolutional Decoder", *IEEE Trans. Commun.*, vol. COM-40, pp. 1810-1819, December 1992.

[Eld59] R. D. Eldred, "Test Routines Based on Symbolic Logical Statements", *Journal of the ACM*, 6, pp. 33-36, January 1959.

[Fet90] G. Fettweis, "Parallelisierung des Viterbi Decoders: Algorithmus und VLSI-Architektur", *VDE Fortschrittsberichte, Reihe 10*, vol. 144, Düsseldorf, FRG: VDE-Verlag, 1990.

[Fet91] G. Fettweis, H. Meyr, "High-Speed Parallel Viterbi Decoding: Algorithm and VLSI-Architecture", *IEEE Communications Magazine*, pp. 46-55, May 1991.

[For73] G. D. Forney, Jr., "The Viterbi Algorithm", *Proc. IEEE*, vol. 61, pp. 268-278, March 1973.

[Gei90] R. L. Geiger, P. E. Allen, N. R. Strader, *VLSI Design Techniques for Analog and Digital Circuits*, McGraw-Hill Publishing Company, New York, 1990.

[Gul86] P. G. Gulak, E. Schwedyk, "VLSI Structure for Viterbi Receivers: Part I—General Theory and Applications", *IEEE J. Selected Areas in Communications*, vol. SAC-4, pp. 142-154, January 1986.

[Gul88] P. G. Gulak, T. Kailath, "Locally Connected VLSI Architectures for the Viterbi Algorithm", *IEEE J. Selected Areas in Communications*, vol. SAC-6, pp. 527-537, April 1988.

[Hag88] J. Hagenauer, "Rate-Compatible Punctured Convolutional Codes (RCPC Codes) and their Applications", *IEEE Trans. Commun.*, vol. COM-36, pp. 389-400, April 1988.

[Hay88] S. Haykin, *Digital Communications*, John Wiley & sons, New York, 1988, pp. 393–414.

[Ish87] T. Ishitani, K. Tansho, N. Miyahara, S. Kubota, S. Kato, "A Scarce-State-Transition Viterbi-Decoder VLSI for Bit Error Correction", *IEEE J. Solid-State Circuits*, vol. SC-22, pp. 575-581, August 1987.

[Iva92] A. Ivanov, S. Pilarski, "Performance of Signature Analysis: A Survey of Bounds, Exact, and Heuristic Algorithms", *Integration, the VLSI Journal 13*, Elsevier Science Publishers B. V., 1992, pp. 17-38.

[Iva93] A. Ivanov, Y. Zorian, "Programmable Space Compaction for BIST", *Symposium on Fault Tolerant Computing*, pp. 340-349, Toulouse, France, June 1993.

[Jir87] Y. Ji-Ren, I. Karlsson, C. Svensson, "A True Single-Phase-Clock Dynamic CMOS Circuit Technique", *IEEE J. Solid-State Circuits*, vol. SC-22, pp. 899-901, October 1987.

[Kal90] S. Kallel, D. Haccoun, "Generalized Type II Hybrid ARQ Scheme Using Punctured Convolutional Coding", *IEEE Trans. Commun.*, vol. COM-38, pp. 1938-1946, November 1990.

[Kal93] S. Kallel, "Efficient Hybrid ARQ Protocols with Adaptive Forward Error Correction", *IEEE Trans. Commun.*, in press.

[Kam93] T. Kameda, S. Pilarski, A. Ivanov, "Notes on Multiple Input Signature Analysis", *IEEE Trans. Computers*, vol. C-42, pp. 228-234, February 1993.

[Kat92] M. Katoozi, A. W. Nordsieck, "Built-In Testable Error Detection and Correction", *IEEE J. of Solid-State Circuits*, vol. SC-27, pp. 59-66, January 1992.

[Lei92] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, San Madeo: CA, Morgan Kaufmann Publishers, Inc., 1992, pp. 730-742.

[Li87] Y. K. Li, J. P. Robinson, "Space Compression Methods with Output Data Modification", *IEEE Trans. on CAD*, vol. CAD-6, pp. 290-294, March 1987.

[Mag90] C. Magerle, "Implementierung des Viterbi Algorithmus", Diplomarbeit an der technischen Universität Wien, 1990.

[Pel91] D. Pellerin, M. Holley, *Practical Design Using Programmable Logic*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1991.

[Qua93] "Q1650C Viterbi Decoder" product description, Qualcomm Inc. 10555 Sorrento Valley Rd., San Diego, CA 92121, USA, 1993.

[Rad81] C. M. Rader, "Memory Management in a Viterbi Decoder", *IEEE Trans. Commun.* vol. COM-29, pp. 1399-1401, September 1981.

[Raj91] J. Rajski, J. Tyszer, "Experimental Analysis of Fault Coverage in Systems with Signature Registers", *Proc. 2nd European Test Conference*, Munich, Germany, April 1991, pp. 45-51.

[Ros93] J. Rose, A. Gamal, A. Sangiovanni-Vincentelli, "Architecture of Field-Programmable Gate Arrays", *Proceedings of the IEEE* vol. 81, July 1993, pp. 1013-1029.

[Shu91] C. B. Shung, P. H. Siegel, H. K. Thapar, R. Karabed, "A 30-MHz Trellis Codec Chip for Partial-Response Channels", *IEEE J. Solid-State Circuits* vol. SC-26, pp. 1981-1987, December 1991.

[Shu90] C. B. Shung, P. H. Siegel, G. Ungerböck, H. K. Thapar, "VLSI Architectures for Metric Normalization in the Viterbi Algorithm", in Proc *Int. Conf. Commun.*, Atlanta, GA, April 1990.

[Sny83] J. S. Snyder, "High-Speed Viterbi Decoding of High-Rate Codes", *Proc. 6th Int. Conf. Digital Satellite Communications*, pp. XII-16-XII-23, Phoenix, AZ, September 1983.

[Sta87] J. Stahl, H. Meyr, "A Bit Serial Viterbi Decoder Chip for the MBit/s Range", *IEEE Custom Integrated Circuits Conference*, pp. 551-554, 1987.

[Sta91] "STEL-2040, STEL-2030A" Product of Stanford Telecom, Santa Clara, CA 95054–1298, USA, Commun. Magazine December 1991.

[Tsu93] B. Tsuji, "On Selecting Programmable Space Compactors for Built-In Self-Test Using Genetic Algorithms", *Master's Thesis*, University of British Columbia, 1993.

[Tri93] S. Trimberger, "A Reprogrammable Gate Array and Applicatins", *Proceedings of the IEEE* vol. 81, July 1993, pp. 1030-1041.

[Vit79] A. J. Viterbi, J. K. Omura, *Principles of Digital Communications and Coding*, New York: McGraw-Hill, 1979, pp. 259-260.

[Wes88] N. H. E. Weste, K. Eshraghian, *Principles of CMOS VLSI Design: A System Perspective*, Addison-Wesley Publishing Co., Reading MA, 1988, pp. 317-319.

[Wu92] Y. Wu, A. Ivanov, "A Fuzzy Multiple Signature Analysis Scheme for BIST", *Proc. 1st Asian Test Symposium*, Japan, November 1992, pp. 247-252.

[Wu93] Y. Wu, A. Ivanov, "Minimal Hardware Multiple Signature Analysis for BIST", *Proc. 11th IEEE VLSI Test Symposium*, Atlantic City, NJ, April 1993, pp. 17-20.

[Wu93a] Y. Wu, "On Multiple Intermediate Signature Analysis for Built-In Self-Test", *Ph.D. Thesis*, University of British Columbia, 1993.

[Xil92] Xilinx, Incorporated, *The Programmable Gate Array Data Book*, Xilinx, Inc., San Jose, CA, 1992.

[Yas81] Y. Yasuda, Y. Hirata, A. Ogawa, "Optimum Soft Decision for Viterbi Decoding", *Proc. 5th International Conference of Digital Satellite Communications*, pp. 251-258, Genoa, Italy, March 1981.

[Yas83] Y. Yasuda, Y. Hirata, K. Nakamura, S. Otani, "Development of Variable-Rate Viterbi Decoder and its Performance Characteristics", *Proc. 6th Int. Conf. Digital Satellite Communications*, pp. XII-24-XII-31, Phoenix, AZ, September 1983.

[Yas84] Y. Yasuda, K. Kashiki, Y. Hirata, "High-Rate Punctured Convolutional Codes for Soft Decision Viterbi Decoding", *IEEE Trans. Commun.*, vol. COM-32, pp. 315-319, March 1984.

Appendix A. Cell Layouts

Simulations are carried out in HSPICE^{TM} for 1.2μm CMOS technology. The ambient temperature is 25° C. V_{DD} = 5 V. All rise and fall times are measured when simulating the cell with the maximum load that a cell has to drive. The rise and fall time here is defined as the time between the input crossing the 2.5 V mark and the output of the load crossing the 2.5 V mark. Inputs are driven by waveforms which have 0.5 ns rise and fall times.

Exclusive Or

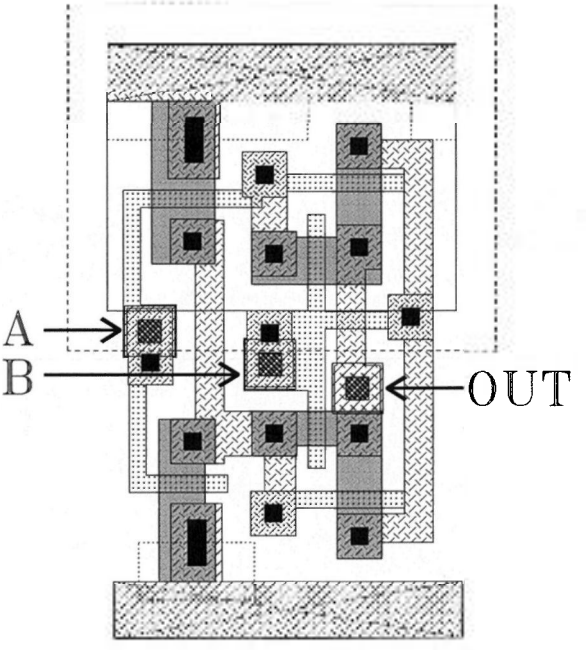


Figure A.1 XOR layout.

Silicon Area: 1000 μm^2.
Rise time: 0.3 ns.
Fall time: 0.4 ns.
Load: 0.05 pF.

D-Flip-Flop

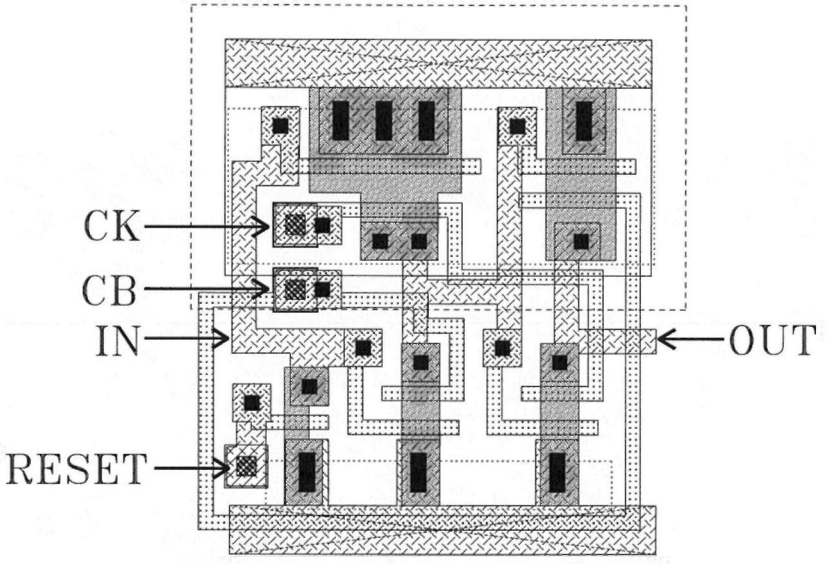


Figure A.2 Layout of a resetable D-flip-flop.

The long wire in poly-Si connecting the two transistors with CB is outside the cell, but when the flip-flops are connected to a shift register, CB will feed the p-channel transistor of the previous stage and the n-channel transistor of its own stage, making the poly-Si loop unnecessary.

Silicon Area: 1470 μm^2.
Rise time: 0.4 ns.
Fall time: 0.3 ns.
Load: 0.04 pF.

Path Memory Cell

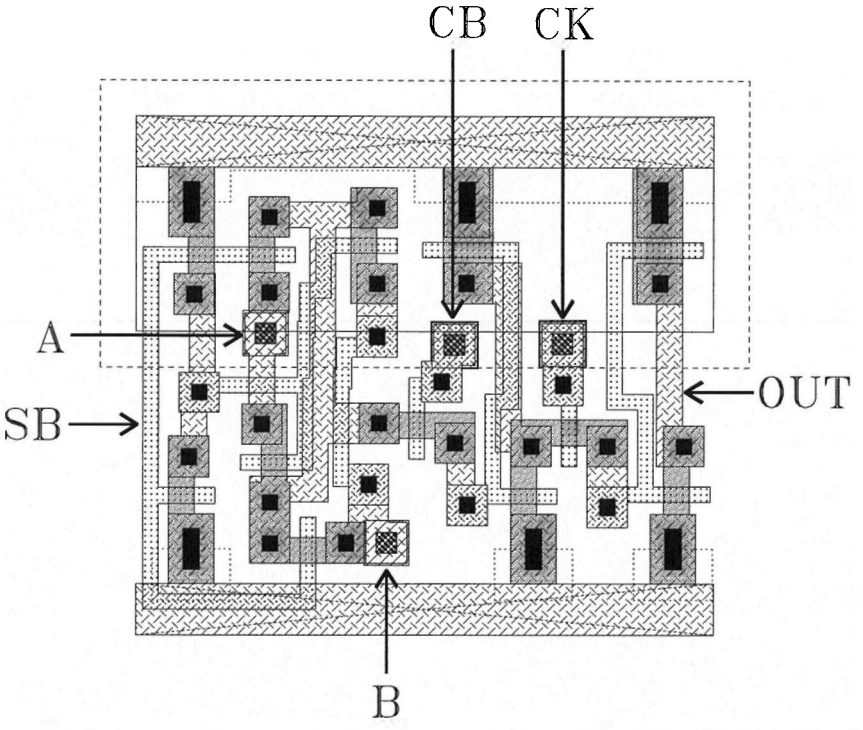


Figure A.3 Layout of a path memory cell, consisting of a multiplexer on the left and a dynamic D-flip-flop on the right.

Silicon Area: 1813 μm^2.
Rise time: 0.3 ns.
Fall time: 0.8 ns.
Load: 0.04 pF.

1-Bit Adder

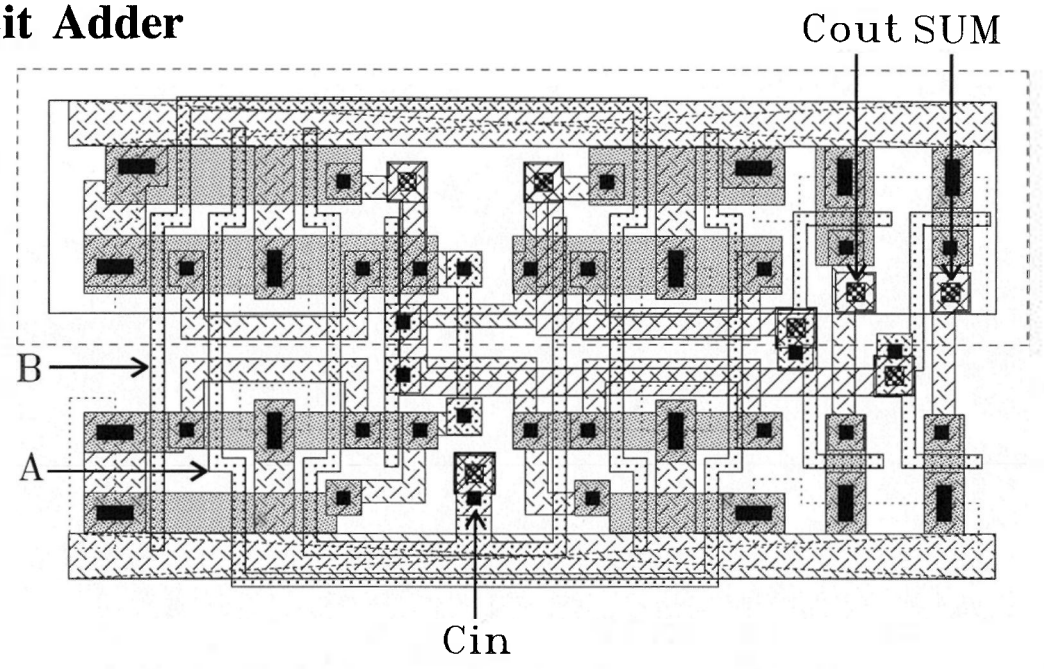


Figure A.4 Adder layout.

Silicon Area: 3494 μm^2.
Rise time SUM: 1.7 ns.
Fall time SUM: 1.9 ns.
Rise time CARRY: 1.2 ns.
Fall time CARRY: 1.1 ns.
Load: 0.06 pF.

Multiplexer

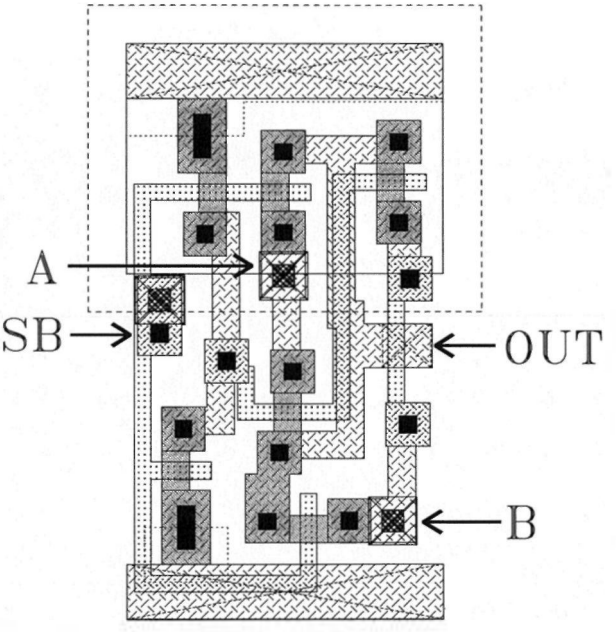
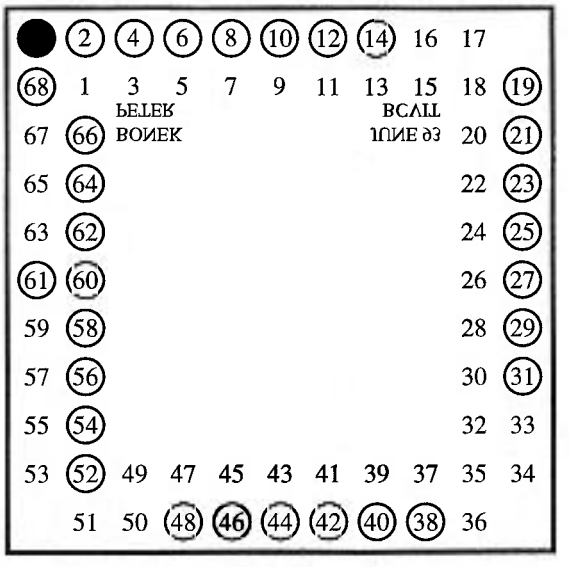


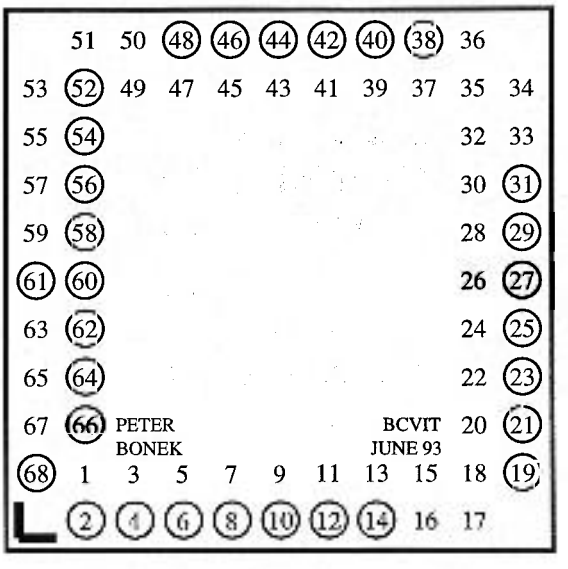
Figure A.5 Multiplexer layout.

Silicon Area: 970 μm^2.
Rise time: 0.6 ns.
Fall time: 0.5 ns.
Load: 0.04 pF.

Appendix B. Pin Locations and Bonding Diagram



Bottom View of Chip



Top View of Chip

Figure B.1 Pin location of the chip on a 68-pin PGA.

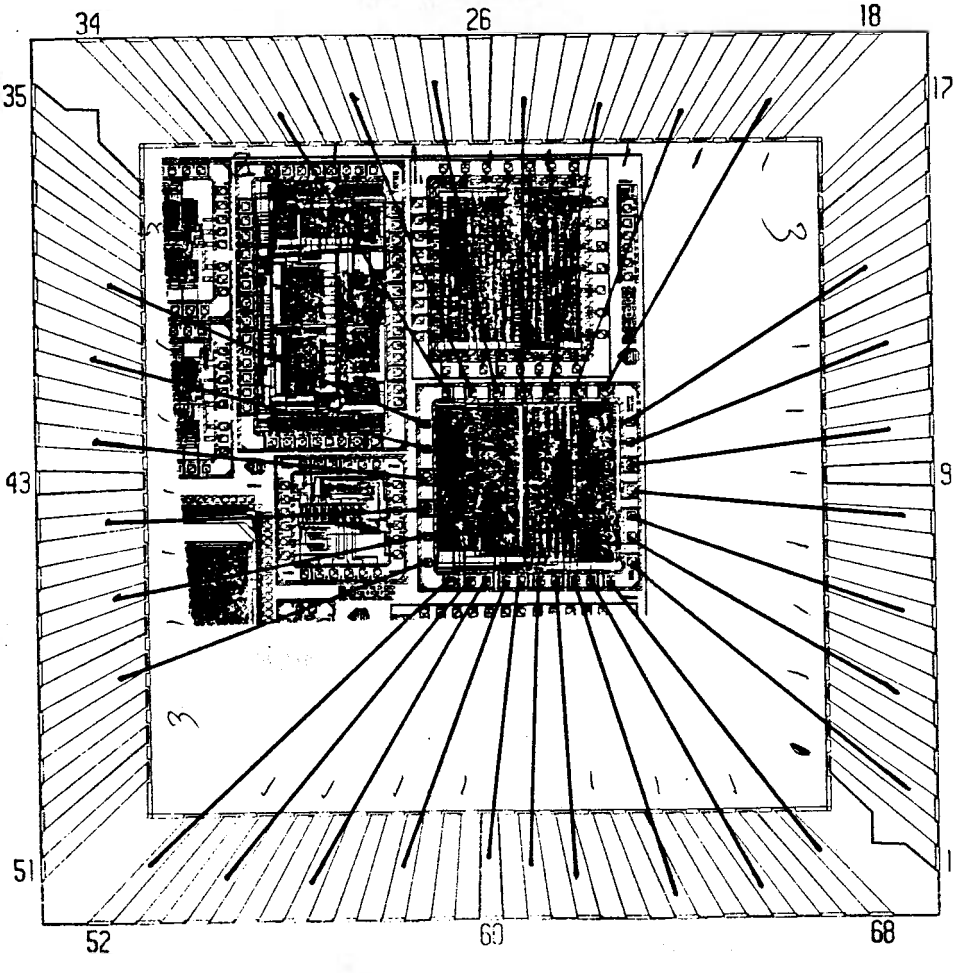
CMC MULTIPROJECT BONDING DIAGRAM BRAND ID LABEL LCA BCVIT
RETICLE CODE V67AIA
OTHER IDENTIFICATION FEATURES
WAFER NUMBERS
DESIGN FILE REFERENCE

PACKAGE CPGA68CC LID C-B34
no substrate connections required, all pins should be floating

WIRE ALLOY	DIA.	ELONG.	T.S.
99% Al/1% Si	.001"	1.5 - 4%	14-16 gms
99% Al/1% Si	.00125"	1.5 - 4%	18-22 gms

D/A PREFORM ALLOY 98% Au/2% Si RECOMMENDED SIZE W/B METHOD U.S.

BONDING DIAGRAM NOTES: 1. DIE ATTACH PAD SIZE: .400 X .400
2. ZERO GROUND



Quantity 5 packages req'd.

ICA V67AIA BCVIT

Figure B.2 Bonding diagram to match the pads on the silicon chip to the pins of the PGA package.

Appendix C. List of Acronyms

ACR	Adaptive Coding Rate
ACS	Add-Compare-Select
ACSU	Add-Compare-Select Unit
AIR	Adaptive Incremental Redundancy
ARQ	Automatic-Repeat-Request
ASIC	Application Specific Integrated Circuit
BER	Bit-Error Rate
BIST	Built-In Self-Test
BM	Branch Metric
BMU	Branch Metric Unit
bmw	branch metric word width
BPSK	Binary Phase Shift Keying
BSC	Binary Symetric Channel
CLB	Configurable Logic Block
CMOS	Complementary Metal Oxide Silicon
CPU	Central Processing Unit
CUT	Circuit Under Test
DMC	Discrete Memoryless Channel
FEC	Forward Error Correction
FILO	First-In Last-Out
FMS analysis	Fuzzy Multiple Signature analysis
FPGA	Field Programmable Gate Array
I/O	Input/Output
IC	Integrated Circuit

LFSR	Linear Feedback Shift Register
LSB	Least Significant Bit
MINSR	Multiple Input No-Feedback Shift Register
MISR	Multiple Input Shift Register
MS	Multiple Signature
MSB	Most Significant Bit
NMOS	N-type MOS
PGA	Pin Grid Array
PLA	Programmable Logic Array
PLD	Programmable Logic Device
PM	Path Metric
PMOS	P-type MOS
pmw	path metric word width
PSC	Programmable Space Compactor
QPSK	Quadrature Phase Shift Keying
PROM	Programmable Read Only Memory
RAM	Random Access Memory
RCC Codes	Rate Compatible Convolutional Codes
RCPC Codes	Rate-Compatible Punctured Convolutional Codes
RCRC Codes	Rate-Compatible Repetition Convolutional Codes
ROM	Read Only Memory
SNR	Signal-to-Noise Ratio
SRAM	Static Random Access Memory
SS	Single Signature
TPG	Test Pattern Generator
VLSI	Very Large Scale Integration
XOR	Exclusive OR