DESIGN METHODOLOGY FOR ANALOG VLSI IMPLEMENTATIONS OF ERROR CONTROL DECODERS

by

Jie Dai

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Electrical Engineering

Electrical and Computer Engineering

The University of Utah

December 2002

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Jie Dai

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

	Chair: Chris J. Myers
	Reid R. Harrison
	Christian Schlegel
	Erik Brunvand
	Gil Shamir

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the dissertation of Jie Dai in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Chris J. Myers
Chair, Supervisory Committee

Approved for the Major Department

V. John Mathews
Chair/Dean

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

In order to reach the Shannon limit, researchers have found more efficient error control coding schemes. However, the computational complexity of such error control coding schemes is a barrier to implementing them. Recently, researchers have found that bio-inspired analog network decoding is a good approach with better combined power/speed performance than its digital counterparts. However, the lack of CAD (computer aided design) tools makes the analog implementation quite time consuming and error prone. Meanwhile, the performance loss due to the nonidealities of the analog circuits has not been systematically analyzed. Also, how to organize analog circuits so that the nonideal effects are minimized has not been discussed.

In designing analog error control decoders, simulation is a time-consuming task because the bit error rate is quite low at high SNR (signal to noise ratio), requiring a large number of simulations. By using high-level VHDL simulations, the simulation is done both accurately and efficiently.

Many researchers have found that error control decoders can be interpreted as operations of the sum-product algorithm on probability propagation networks, which is a kind of factor graph. Of course, analog error control decoders can also be described at a high-level using factor graphs. As a result, an automatic simulation tool is built. From its high-level factor graph description, the VHDL simulation files for an analog error control decoder can be automatically generated, making the simulation process simple and efficient.

After analyzing the factor graph representations of analog error control decoders, we found that analog error control decoders have quite regular structures and can be built by using a small number of basic cells in a cell library, facilitating automatic synthesis. This dissertation also presents the cell library and how to automatically synthesize analog decoders from a factor graph description.

All substantial nonideal effects of the analog circuit are also discussed in the dissertation. How to organize the circuit to minimize these effects and make the circuit optimized in a combined consideration of speed, performance, and power is also provided.

To my family.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the support and help from many people. First, I owe my deepest gratitude to my advisor, Chris Myers, for introducing me into the field of analog implementation of error control coding systems. He has been very patient, especially during my early period of time at the University of Utah, when I had to learn many things from the ground. His guidance and encouragement helped me go through the most difficult time of my research when I felt lost and restless. I also would like to thank Christian Schlegel and Reid Harrison for their technical suggestions and comments on this dissertation and for serving on my supervisory committee. My thanks also extend to Erik Brunvand and Gil Shamir for serving on my supervisory committee.

I would like to thank my fellow students, Christopher Winstead, Shuhuan Yu, and Scott Little, for their contributions to my research work. Special thanks to Christopher Winstead and Shuhuan Yu for the time they invested in long and profound discussions on many different topics. I would also like to thank my officemates, Eric Mercer, Eric Peskin, Kip Killpack, Curt Nelson, and Yanyi Zhao, for their help so that I could focus on my research.

The love, advice, encouragement, and support from my family were essential in completing this dissertation. During my PhD study, my parents made me believe that obtaining the PhD is not only their dream for me, but also one of the most important and prestigious milestones a person could reach. My wife has always been there with me, providing so much help so that I could concentrate on my research, encouraging me and giving me invaluable advice when things did not go well, and celebrating with me when I made progress.

This research is supported by NSF grant CCR-9971168.

CHAPTER 1

INTRODUCTION

In 1948, Shannon proved that it is possible, by proper *encoding* of the information source, to reduce errors induced by a noisy channel to any desired level, without sacrificing the rate of information transmission or storage of a given channel, as long as the rate is below the so-called *channel capacity* [61]. The term encoding in this context means that redundant information is added to the data stream. Since then, his theory has been called *the Shannon limit*. In the last few years the demand for efficient and reliable digital data transmission and data storage has tremendously increased. As a result, researchers have found more efficient coding schemes to approach the Shannon limit. However, in general, we can state that the more complex our coding schemes are constructed, the more protection we get from coding. On the other hand, decoding has become more complicated. The computational complexity of decoding for codes that try to reach the Shannon limit grows more than linearly, i.e., quadratically or even exponentially. Today's state-of-the art codes such as *Turbo codes*, *low-density parity-check codes*, and other similarly built codes need huge computational power to deliver real-time results.

In order to meet the rapidly growing computational effort, processing speed has to be boosted by using more sophisticated semiconductor processes. Unfortunately, unless more parallelism is introduced in the decoding system, the processing speed just increases linearly with the clock frequency. Also, as the complexity and processing speed increases, the power consumption increases dramatically, which makes the decoding strategy not practical for system consideration and not appropriate for mobile devices.

Recently, researchers have found that bio-inspired analog network-decoding is a good approach. Mead's outstanding work on neuromorphic systems has clearly shown that neuromorphic systems can reach outstanding precision on the system level [52]. Similarly, analog decoders also reach a quite good performance at the system level [67] [26] [42] [69]. Compared with its digital conterparts, analog decoders have a much better combined power/speed performance. Also, researchers have observed that a number of important

algorithms in error-control coding can be interpreted as operations of the *sum-product algorithm* on *probability propagation network*, which is a kind of *factor graph* [17] [35] [2]. The sum-product algorithm on probability propagation networks can be implemented using analog VLSI [67] [26] [38]. As a result, analog decoders have been built using BiCMOS [54] and recently by our research group using CMOS [69].

In designing our CMOS Hamming decoder, the issue of how to simulate this analog circuit arose. This problem is even more complex when we must determine the error rate at different noise levels efficiently and accurately. The error rate at a low noise level is so low that a huge number of simulations must be done before we can know the error rate. Of course, how to simulate the analog circuit efficiently becomes the dominant problem. Although `Spice` is an accurate circuit simulator, it is mainly aimed at the transistor level and is too time consuming for large circuit simulations. As a result, high level simulation must be used.

Because an analog error control decoder can be represented by its factor graph, it is possible to simulate it at the factor graph level. As a result, an atomatic simulation tool can be built. The tool accepts a factor graph representation of an analog error control decoder and generates the VHDL simulation file needed to do simulation. Thus the simulation process can be made easy and quick.

Also, we noticed that the factor graph representations for error control coding systems are always quite regular. Therefore, the corresponding circuits can be constructed by duplicating only a few fundamental circuit cells. As a result, if a library including the needed circuit cells has been built, by using files describing the factor graph of the system, a circuit implementation can be generated. Automatic synthesis and automatic simulation at a circuit level for analog circuits are difficult because analog design is usually less systematic and more heuristic compared with digital design. However, error control decoders are systematic, using only a few basic cells to build the whole circuit. This makes automatic synthesis and simulation at the circuit level for such kinds of circuits quite practical.

In designing the Hamming decoder, we also noticed many design issues. For example, mismatch, noise, and channel length modulation can all affect the functionality of the circuit. Also, researchers have found that CMOS circuits working under strong inversion can still make analog decoders work with degraded performance. In conclusion, the nonidealities of the analog circuit can affect the performance. These nonidealities need

to be modeled and simulated. Using these simulation results, a strategy for designing analog decoders to minimize these nonideal effects is developed.

1.1 Related Work

Frey [17], Kschischang [35], and Aji [2] have observed that a number of important algorithms in error-control coding can be interpreted as operations of the sum-product algorithm on probability propagation networks, which are a kind of factor graph. Wiberg [67], Hagenauer [26] and Loeliger [38] have noticed that the sum-product algorithm on probability propagation networks is well suited for analog VLSI with exponential current-voltage relationships. Moerz [54] and Lustenberger [42] have built analog decoders by using BiCMOS. Recently, our group has built an analog CMOS Hamming decoder [69].

Gielen [20] and other researchers have investigated computer-aided design of analog and mixed-signal integrated circuits. In general, they have concluded that there were not yet any robust commercial CAD tools to support or automate analog circuit design apart from circuit simulators. Some of the main reasons for this lack of automation are that analog design in general is perceived as less systematic and more heuristic and knowledge-intensive in nature than digital design, and that it has not yet been possible for analog designers to establish a high level of abstraction that shields all the device-level and process-level details from the high level design. For simulation and modeling, VHDL-AMS seems quite promising. However, it is still too time consuming for analog error control decoder simulation and there is little CAD tool support [13] . For automatic synthesis, tools are either ad-hoc heuristic methods such as `OASYS` [28], `BLADES` [14], `OPASYN` [33] or using optimization-based approaches such as [51] that still need plenty of work.

In designing the analog QCRA decoder, Lustenberger found that the decoder has a regular structure and the entire circuit can be built by using a few basic cells. As a result, he built a few basic cells and then used a tool to automate the construction of the decoder using these cells [42]. However, using his method, different decoders may need different basic cells. As a result, it is still not a general method that can be used for different analog error control decoders. Our research shows that all analog error control decoders can be built by using a small number of basic cells in a cell library, facilitating automatic synthesis. Moreover, our analysis shows that analog error control decoders built using this method are comparable in performance, smaller, and lower power than the corresponding *canonical designs*. Also, using the basic cells to construct the circuit

makes automatic simulation at the circuit level possible.

Lustenberger has investigated the mismatch effect and shows a simulation technique for mismatch [45]. However, how much the decoder's performance is affected by mismatch and how to organize the decoder to minimize the mismatch effect is not discussed and left for time-consuming Monte-Carlo simulation to solve.

Lustenberger also shows in simulation that CMOS circuits working under the strong inversion region can still make analog decoders work [42]. However, the reason this is true has not been studied.

1.2 Contributions

In the past few years, researchers have found that using analog circuits to design error control decoders is a promising approach. However, no one has systematically investigated the modeling and implementation issues. Also, no researchers have provided a general method to automate the design process. This thesis provides a design methodology for analog implementation of error control decoders that solves these problems. There are four major contributions of this thesis.

The first contribution is the development of a high-level simulation method using standard VHDL. Because a large amount of the simulation task is to get the *bit-error rate (BER)* curve, a high level simulation method is needed. VHDL is quite efficient in doing simulation of parallel architectures so we use it to do the high level simulation. Although VHDL-AMS can also be used to do simulation, it is still too complicated to be efficient. By using real values, VHDL is used to do simulation efficiently and accurately.

The second contribution is the automatic simulation method. Because error control decoders can be described by their factor graph description, a tool is built to generate VHDL simulation files from the factor graph description of the analog error control decoder. This greatly reduces the work to generate simulation files and helps speed up the design exploration process.

The third contribution is the automatic synthesis method, which greatly speeds up the design process for analog error control decoders. We have discovered that all analog decoders could be partitioned into small circuit cells where the total number of these circuit cells that is needed to build current analog decoders is limited. As a result, an automatic synthesis tool that enables automatic synthesis of analog decoders from its factor graph description is built. The cell library and the automatic synthesis tool greatly

speed up the design process of analog decoders, which is time consuming, especially for the state-of-art analog decoders that are quite large.

The fourth contribution is in the area of modeling and implementation issues. The nonidealities of the analog circuit are discussed. Methods to model these nonidealities are provided and techniques to minimize effects of these nonidealities are developed. In general, the organization of an analog decoder that is optimized in a combined consideration of speed, performance, and power is also provided.

1.3 Design Flow

Figure 1.1 shows the design flow for automatic synthesis and simulation. The factor graph files (.fg files) for the decoder provide the factor graph description of the decoder that is quite simple. Then, using these factor graph files, the compiler generates a high-level VHDL description file for the decoder. In order to do simulation to find out the bit error rate, the encoder should also be provided as a simulation environment file. Using the simulation environment file, the compiler generates the VHDL simulation environment file. Using the high-level VHDL description of the decoder and the VHDL simulation environment file, a high-level VHDL simulation can be done to verify whether the structure of the decoder is correct or not. Using the high-level VHDL description of the decoder and the cell library, the high level VHDL description of the decoder can be further decomposed into a cell library based VHDL structural description. The cell library based VHDL structural description together with the layout of the cell library can be used by an automatic layout generation tool to generate layout of the decoder automatically. Also, the cell library based VHDL structural description together with the circuit parameters of the cell library can be used to generate a circuit level simulation result. Because the cell library needs to be built only once, using the automatic synthesis and simulation technique, only the factor graph description of the decoder and its simulation environment description is needed to simulate and design an analog decoder. Of course, the simulation and design process is greatly sped up.

1.4 Dissertation Outline

This dissertation is organized as follows: Chapter 2 gives some background information about communication systems, coding theory, and the channel models that are essential to understand this dissertation.

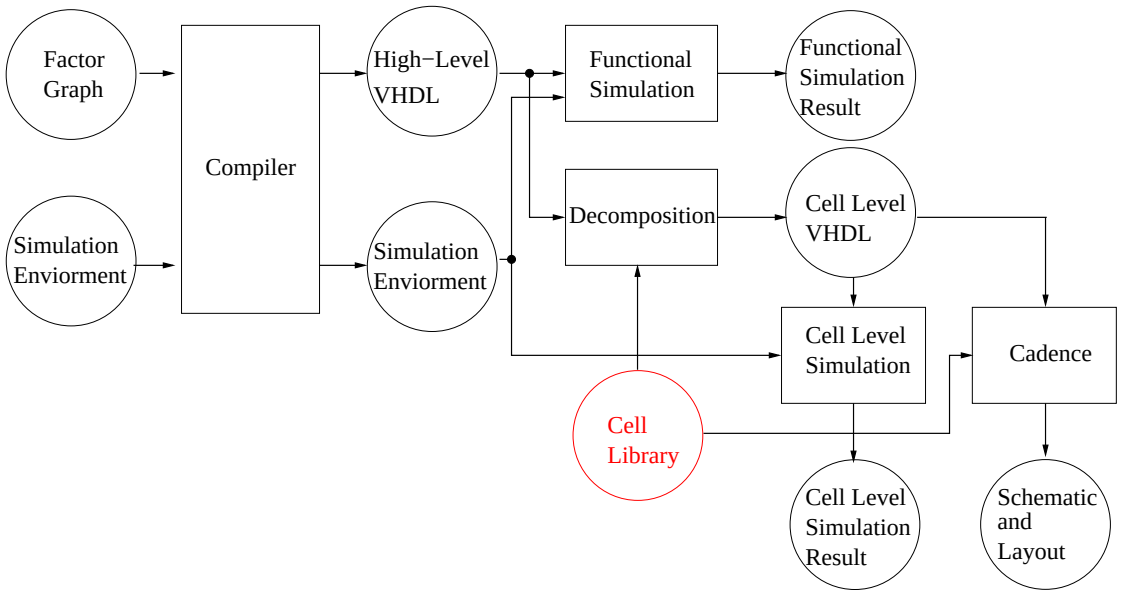

Figure 1.1. Data flow for automatic synthesis and simulation.

Chapter 3 describes factor graphs and the sum product algorithm, which is the basis for automatic simulation and synthesis. This chapter also describes an automatic simulation method that generates VHDL simulation files from an analog error control decoder's factor graph description.

Chapter 4 first describes translinear circuits, which is the basis of circuit implementation. Then it describes the probability calculus modules and how the probability calculus modules are constructed and how to use the probability calculus to construct analog decoders. This chapter also describes how to partition probability calculus modules into small cells and how to organize these small cells into a cell library that enables automatic synthesis of all current practical analog decoders. At last, this chapter describes how to do automatic synthesis from the factor graph description of an analog decoder.

Chapter 5 includes many circuit implementation and simulation issues. The modeling and simulation issues of device mismatch, internal noise, thermal effects, and channel length modulation are discussed in this chapter. Then, MOS circuits working in the strong and moderate inversion region are discussed. The performance loss due to quadratic behavior is also discussed. A one pole system approximation is provided to enable circuit level simulation. At last, the technique of automatic generation of circuit level simulation is provided.

In Chapter 6, some case study results are included for an (8,4) Hamming decoder and a $(16, 11)^2$ Hamming product decoder.

Chapter 7 summarizes this dissertation, and discusses the future work.

CHAPTER 2

BACKGROUND INFORMATION

This chapter briefly describes some basic information about error control coding systems. To simplify the description, we restrict ourselves to the binary case.

2.1 Parity-Check

In communication, the information transmitted can be deteriorated by noise. As a result, the received information may not be the information that is transmitted. In order to overcome this problem, when we transmit the information, we add some redundancy. Then, when the receiver receives the information, it checks whether the information has been deteriorated by noise and attempts to determine the information transmitted. For example, just by simply repeating the information that is transmitted twice, redundancy is added and the receiver can check whether the information has been deteriorated or not. Actually, in the natural language, plenty of redundancy is used and we call it context. Of course, by adding redundancy, more information needs to be transmitted and received. In general, the more redundancy we use, the more protection we get from the redundancy. However, we pay more for transmitting and receiving redundancy. In order to measure how much redundancy is added, a term *rate* is used. It equals the number of information bits divided by the number of bits that are sent out. Equation 2.1 shows the definition of rate in which k is the number of information bits and n is the number of bits that are sent out.

$$R = k/n \tag{2.1}$$

Of course, we would like to add the smallest amount of redundancy to get the maximum protection from the redundancy. Such a problem is the encoding and decoding problem. In 1948, Shannon proved that it is possible, by proper encoding of the information source, to reduce errors induced by a noisy channel to any desired level, without sacrificing the rate of information transmission or storage of a given channel, as long as the rate is below

the so-called channel capacity. However, Shannon did not point out how to realize it. Such a problem is left for the encoding and decoding strategy to solve and researchers have spent several decades to reach the Shannon limit. In the binary field, it is widely known that by adding an additional *parity bit* to make the sum of bits to be always '1' or '0', we can easily find out whether an error has happened or not assuming only 1 bit error happens. Assuming that we are using even *parity-check*, then Equation 2.2 is true (In binary field GF(2), addition is mod 2 or XOR and multiplication is AND). The redundant bit x_{k+1} is called the parity-check bit and Equation 2.2 is called the parity-check equation. If we write down the coefficients of the parity-check equation as a row vector, we can have a matrix that has only one length $k+1$ row vector as shown in Equation 2.3. However, we only know whether one error exists or not when only one parity-check bit is used assuming at most one bit error can happen. Can we find more errors and even correct the errors by adding more parity-check bits and parity-check equations?

$$x_1 + ... + x_k + x_{k+1} = 0 \tag{2.2}$$

$$\begin{bmatrix} 1 & 1 & \cdots & 1 \end{bmatrix} \tag{2.3}$$

2.2 Basic Coding Concept

Before answering this question, we would like to introduce some basic coding concepts in the following subsections.

2.2.1 Coding System

Figure 2.1 shows a simplified communication model used by many textbooks [48] [37] [62]. In Figure 2.1, u is the source code, x is the output of the encoder, y is the received value after transmission through the coding channel in which the encoded output x is deteriorated by noise n, and $\hat{u}$ is the output of the decoder. These symbols are used in the remaining of this thesis. Also, u_k, x_k, y_k, n_k, and $\hat{u}_k$ are also used to represent the source, encoded output, received value, noise, and decoded output for a particular bit.

2.2.2 Systematic Codes

If the uncoded information bits are transmitted together with the parity-check bits, we call the code *systematic*. Otherwise it is called *nonsystematic*. For example, when there

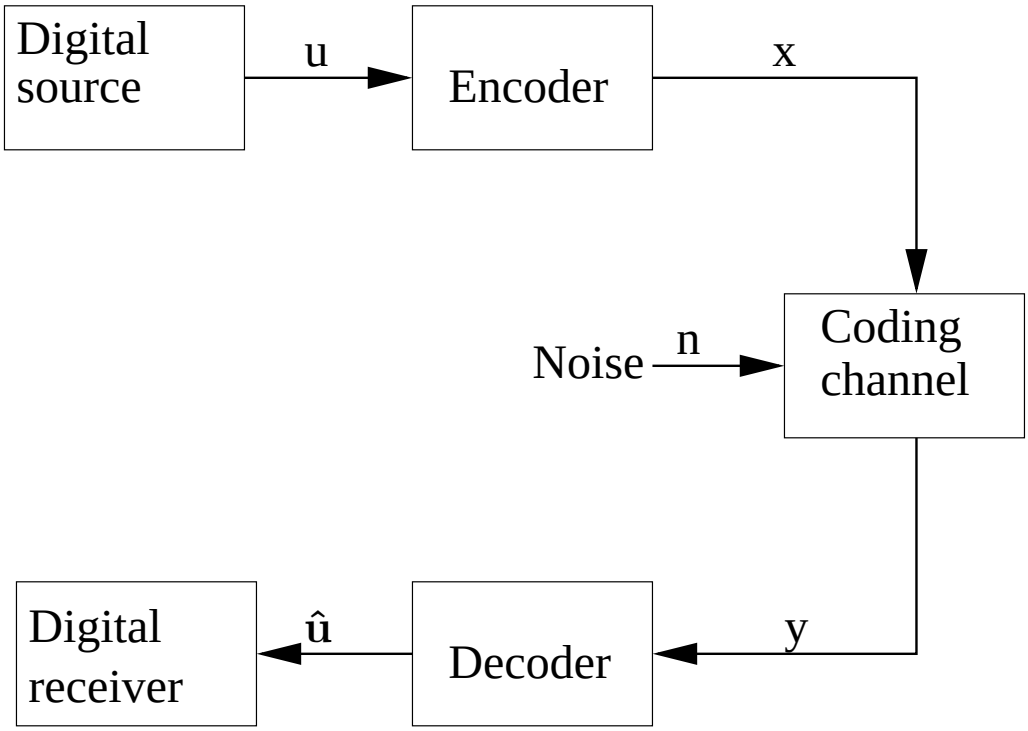

Figure 2.1. Simplified model of a communication system.

are three information bits u_1, u_2, and u_3, if the three information bits are transmitted along with the even parity-check bit $x_4 = x_1 + x_2 + x_3$, then the code is systematic. However, if the information bits are not transmitted, instead the following bits are transmitted, $x_1 = u_1 + u_2$, $x_2 = u_1 + u_3$, $x_3 = u_2 + u_3$, and $x_4 = u_1 + u_2 + u_3$, then the code is not systematic. In general, systematic codes can be more easily coded and decoded than nonsystematic codes so they are more widely used.

2.2.3 Linear Codes

If the sum of two valid codewords is also a valid codeword, the code is called a *linear code*. For example, the even parity-check code is a linear code while the odd parity-check code is not a linear code. Using a code with two information bits and one parity bit as an example, if even parity-check is used, since 011 and 110 are valid codewords, then 101 is also a valid codeword. However, if odd parity-check is used, 001 and 100 are valid codewords while their sum 101 is not a valid codeword.

2.2.4 Hamming Distance

With more parity-check equations, we may find more errors and correct more errors. However, how can we know how many errors a code can detect and how many errors it can correct? Hamming gave the answer by using the concept of *Hamming distance*. The distance between two codewords is defined as the number of positions that they differ. Hamming distance is defined as the minimum distance between any two codewords of the code. If a codeword has a larger distance between any other valid codewords, it is more unlikely for the received codeword to be regarded as another valid codeword. Figure 2.2 shows how to use Hamming distance to find out how many errors a code can detect and how many errors it can correct. Assuming that all the codewords are located in a plane, the valid codewords are located at A, B, C, D, $\cdots$ and Hamming distance is 3. If 1 bit error occurs on A, then the codeword is moved to the circle, which has A as its center and radius 1. If two bit errors occur on A, then the codeword is moved to the circle that has A as its center and radius 2. From Figure 2.2, it is easy to understand that a code with Hamming distance 3 can correct only one error. If a code had Hamming distance d, then it can correct $\lfloor (d-1)/2 \rfloor$ errors where $\lfloor x \rfloor$ denotes the greatest integer less than or equal to x. If d is even, the code can simultaneously correct $(d-2)/2$ errors and detect $d/2$ errors. Because the combinations of two valid codewords is large, it is difficult to find the Hamming distance by using the definition. However, for linear codes, the sum of any

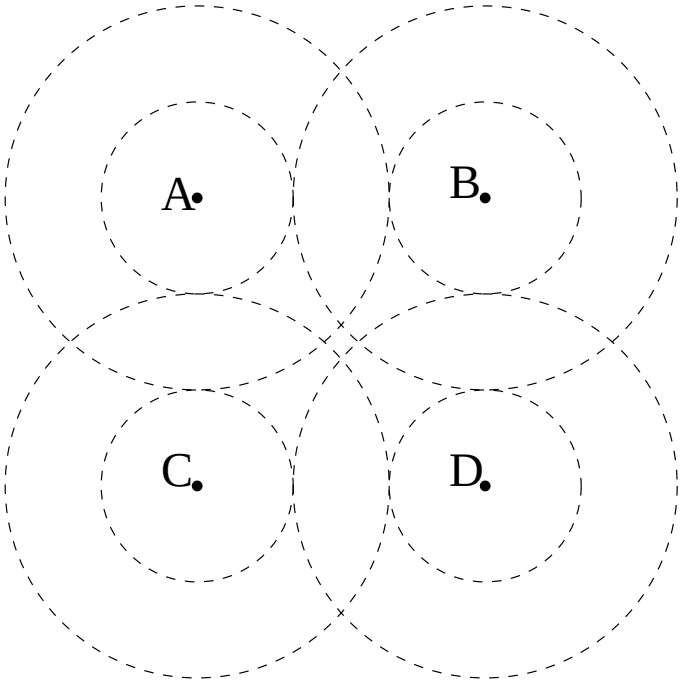

Figure 2.2. The relationship between Hamming distance and error detection, error correction.

two valid codewords is also a valid codeword. As a result, for linear codes, the Hamming distance equals the minimum number of 1's of a valid codeword that is defined as the *Hamming weight*.

2.3 Block Codes

A binary *block code* is defined as an algebraic mapping from the vector space 2^k into the vector space 2^n. Thus, in the vector space 2^n, only 2^k codewords are valid. k is the number of information bits and n is the code length and the code is called an (n, k) block code with rate $R = k/n$. For example, the even parity-check code described by Equation 2.2 is a $(k + 1, k)$ block code with rate $R = k/k + 1$.

If the mapping from vector space 2^k into the vector space 2^n is linear, then the block code is a linear block code, otherwise it is not a linear block code. For a linear block code, given the k bit information sequence u, the n bit codeword x is always determined by multiplying u with a *generator matrix*, G, shown in Equation 2.4. In the equation, u is a length k row vector, x is a length n row vector where G is a $k * n$ matrix. For example, the generator matrix for the even parity-check code shown in Equation 2.2 is shown in

Equation 2.5. However, the odd parity-check code is not a linear block code and there is no generator matrix for the odd parity-check code.

$$x = u \cdot G \tag{2.4}$$

$$G = \begin{bmatrix} 1 & 0 & \cdots & 0 & 1 \\ 0 & 1 & \cdots & 0 & 1 \\ \cdots & \cdots & \cdots & \cdots & 1 \\ 0 & 0 & \cdots & 1 & 1 \end{bmatrix} \tag{2.5}$$

For a linear block code, we can always find a matrix H that satisfies Equation 2.6 where H is a $(n-k)*n$ matrix that is called the *parity-check matrix*. Notice that for linear block codes, all-zero input information bits is transformed into an all-zero codeword. As a result, the right side of Equation 2.6 is a length $n-k$ column vector. For linear block codes, from its generator matrix, we can find out its parity-check matrix and vice versa. The parity-check matrix for the even parity-check code shown in Equation 2.2 has been shown in Equation 2.3.

$$H \cdot x^T = 0^T \tag{2.6}$$

Actually, a linear block code uses even-parity check on part or all of its information bits. As a result, from these equations, its parity-check matrix can be directly written out just as done for the even parity-check code shown in Equation 2.2. Then, from its parity-check matrix, its generator matrix can be found.

In 1950, Hamming provided a class of linear block codes that could correct one error, which we now call *Hamming codes* [27]. If the code length $n = 2^r - 1$, then the parity-check matrix H of the Hamming code is constructed by using all the nonzero length r vectors as the columns of H. Thus a Hamming code is an $(n = 2^r - 1, k = 2^r - r - 1)$ linear block code.

Using a Hamming (7,4) code as an example, $n = 7, k = 4, r = 3$, there are 4 information bits, the codeword length is 7, and 3 even parity-check equations exist. Let's use $x_1, x_2, \cdots, x_7$ to denote the 7 bits in the codeword. If the even parity-check equations shown in Equation 2.7 are used, then the corresponding parity-check matrix is shown in Equation 2.8 and this parity-check matrix was the one Hamming used in his paper [27].

$$
\begin{aligned}
x_4 + x_5 + x_6 + x_7 &= 0 \\
x_2 + x_3 + x_6 + x_7 &= 0 \\
x_1 + x_3 + x_5 + x_7 &= 0
\end{aligned} \tag{2.7}
$$

$$
H = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \tag{2.8}
$$

Notice that the column vectors can have different permutations and the resulting parity check matrix, H, is different. However, the constructed codes all have the same rate and error correction ability and they differ only where the parity-check bits are inserted. As a result, we call them equivalent. Equation 2.9 shows another parity-check matrix of the famous Hamming (7,4) code where I_r is a $r * r$ identity matrix.

$$
H = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} P & I_r \end{bmatrix} \tag{2.9}
$$

If a systematic code is used and all the information bits are sent out before the parity-check bits, then we can have a generator matrix shown in Equation 2.10 where I_k is a $k * k$ identity matrix, and Q is a $k * (n - k)$ matrix.

$$
G = \begin{bmatrix} I_k & Q \end{bmatrix}^T \tag{2.10}
$$

Since every row vector of the generator matrix is a valid codeword, Equation 2.11 can be derived in which 0_r is a $r * r$ all zero matrix.

$$
\begin{aligned}
HG^T &= 0_r \\
\begin{bmatrix} P & I_r \end{bmatrix} \begin{bmatrix} I_k & Q \end{bmatrix}^T &= 0_r
\end{aligned} \tag{2.11}
$$

From Equation 2.11, it is easy to see that $P^T = Q$. As a result, for a linear block code, from its parity-check matrix, which has the form $[PI_r]$, we can immediately find its generator matrix $[I_k P]^T$ and vice versa. Of course, from the parity-check matrix of the Hamming (7,4) code shown in Equation 2.9, we can find the generator matrix shown in Equation 2.12.

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \tag{2.12}$$

From the generator matrix shown in Equation 2.12, we can find that the Hamming weight for the Hamming (7,4) code is 3. As a result, the Hamming distance is also 3 and the Hamming (7,4) code is also called the Hamming (7,4,3) code. Because it has Hamming distance 3, it can correct 1 error.

Notice that if a single error happens, then the value at the right end of Equation 2.6 is not a zero vector. Instead, the result is a column vector of the parity-check matrix and the position of the column vector is the position in which the error happens. If the parity-check matrix of the Hamming (7,4,3) code is written in the form of Equation 2.13, then a single error is identified by h_j and j is the location of the error and all h_j are unique. However, double errors in positions i and j are not identifiable since $h_i + h_j = h_k$ look like a single error in position k.

$$H = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} h_1 & h_2 & h_3 & h_4 & h_5 & h_6 & h_7 \end{bmatrix} \tag{2.13}$$

From the generator matrix shown in Equation 2.12, you can see that all codewords of the Hamming (7,4) code other than the all 0 codeword and the all 1 codeword have weight 3 or 4.

Any code can be extended by adding an overall parity-check equation to ensure that all symbols add up to zero modulo 2. This is done by modifying the parity-check matrix as Equation 2.14 shows where H is the old parity-check matrix.

$$\hat{H} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ & & & 0 \\ & H & & \vdots \\ & & & 0 \end{bmatrix} \tag{2.14}$$

If we choose the parity-check matrix of the Hamming (7,4) code to be the one shown in Equation 2.15, by doing the transformation shown in Equation 2.14, the parity-check matrix shown in Equation 2.16 is derived. By adding line 3 to line 1 and then move the

last column to the first column, we can have another parity-check matrix of the extended Hamming (8,4) code shown in Equation 2.17.

$$H = \begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \tag{2.15}$$

$$H = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix} \tag{2.16}$$

$$H = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \tag{2.17}$$

It is interesting to notice that for the extended Hamming (8,4) code shown in Equation 2.17, $H \cdot H^T = 0_{4*4}$. As a result, every row vector of H is a valid codeword and H is also the generator matrix G for the extend Hamming (8,4) code. From Equation 2.17, we see that all the codewords other than the all 0 codeword and the all 1 codeword have weight 4. As a result, the Hamming distance for this code is 4 and it is called the *extended Hamming (8,4,4) code*.

2.4 Convolutional Codes

For block codes, usually redundancy is generated by the algebraic equations. However, hardware implementation of the algebraic equations is not easy. As a result, the hardware implementation of the encoder and decoder of a block code is not easy. This makes people consider other ways of providing redundancy.

In sequential circuit design, *finite state machines* are widely used and can be easily implemented. If the encoded output can be limited by the current state of a state machine, then redundant information can be added according to the current state. If the states are different, then the redundancy is added differently just like having different parity-check equations in the block code. The number of registers used is called the memory order. In general, the larger the memory order, the larger the state number it can provide and the more protection the code gains. This kind of code is called a *convolutional code*. Figure 2.3 shows an example of a convolutional code and we use this example to explain

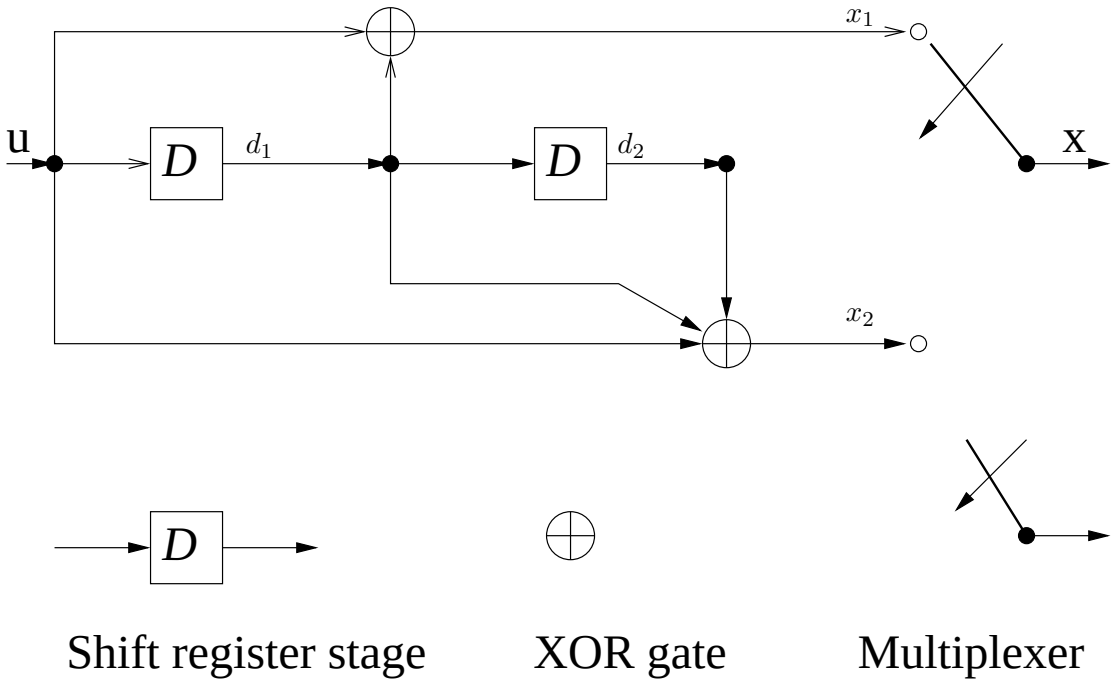

Figure 2.3. Convolutional code example.

some concepts related to convolutional codes. In Figure 2.3, there are two shift registers and the corresponding stored value are d_1, d_2. Every time when an input u comes, the encoder generates two outputs $x_1 = u + d_2$ and $x_2 = u + d_1 + d_2$. As a result, this convolutional code has rate $R = 1/2$. Because there are only two registers, the maximum number of states is 4. If we use $d_1 d_2$ to denote states and use $u/x_1 x_2$ to label state transitions, then we the corresponding state transition diagram is shown in Figure 2.4. For example, if $d_1 = 1, d_2 = 0$ and $u = 0$, then the new values of $d_1 = 0$, $d_2 = 1$ and the output $x_1 = u + d_2 = 0 + 0 = 0$, $x_2 = u + d_1 + d_0 = 0 + 1 + 0 = 1$. As a result, there is a state transition from 10 to 01 with label 0/01 on the transition.

Notice that for convolutional codes, the initial input can affect the next state, thus all the following encoded output could be affected. As a result, the input-output sequence can be infinite, unlike block codes. However, because the state number is finite, at some time in the future, the state must be a state that has been seen before.

It is easy to convince yourself that the decoder of a convolutional code can also be easily built by a finite state machine. In real applications, several continuous '0' bit inputs are always used to terminate the sequence to make the state return to the all '0' state.

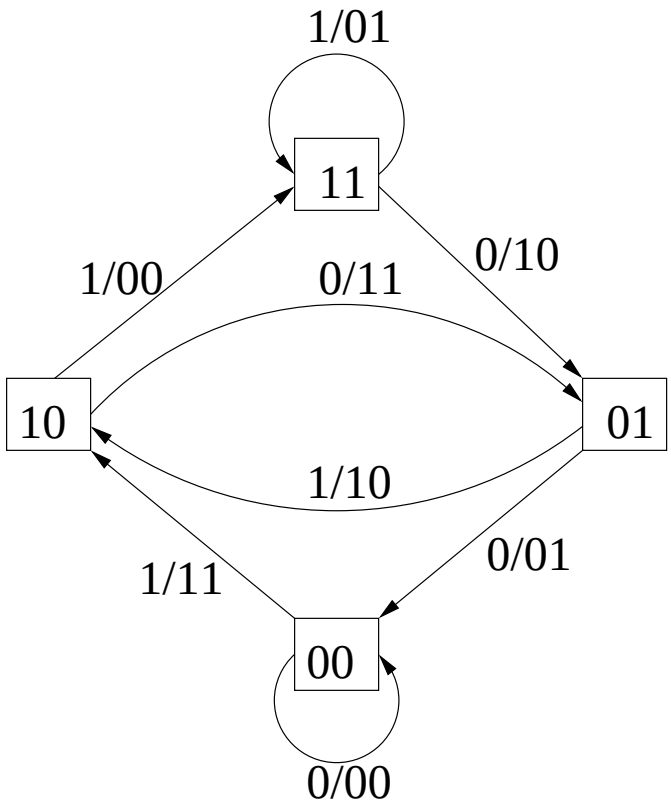

Figure 2.4. The state-transition diagram of the convolutional code from Figure 2.3.

2.5 Trellis Coding

2.5.1 Trellis Coding for Convolutional Codes

If we assume the initial state of the convolutional code shown in Figure 2.3 is '00', by following the state transition diagram and drawing the state as a node and the transition between states as a branch, we can get a tree description of the code shown in Figure 2.5. Each path along the tree represents a valid codeword. However, as we proceed, the tree explodes very quickly. Noticing that the output of time $t+1$ is simply defined by the state of time t and the input, we can represent all nodes showing the same state memory content at a given time by using only one node. As a result, we can get the *trellis* representation of the code shown in Figure 2.6, which is more compact.

From the example, we know that there are only four different states to depict. After encoding u_1 and u_2, the state number reaches the maximum number. Then, when the following information bit comes, the state number does not increase anymore and the state set is the same as the previous state set. Because the next state is only determined by the current state and input, it is easy to understand that the trellis of the code is simply built by concatenating identical trellis sections as shown in Figure 2.7. From the derivation of the trellis, we notice that for a single trellis section, the left nodes show the current state $S(t)$ where the right nodes show the next state $S(t+1)$. The branch between left state and right state shows a valid state transition and combination of encoded input/output.

2.5.2 Trellis Coding for Block Codes

The previous subsection shows that a convolutional code has a trellis representation. Are there trellis representations for block codes? The answer is absolutely yes. Since every trellis path represents a valid codeword, if we use a trellis as a visual method of the code, then every code has its trellis representation and each distinct codeword corresponds to a distinct path through the trellis. Now, the problem is how to find the trellis representation of block codes. Bahl, Wolf, and Massey [4] [71] [50] found the answer. Following the approach in [4] [71], we define the states of the trellis as follows: let s_r be the label of the state at time r and let the initial state s_1 to be in the 0 state that means $s_1 = 0$. Then,

$$s_{r+1} = s_r + x_r h_r = \sum_{l=1}^{r} x_l h_l \tag{2.18}$$

where h_r is the rth column of the parity-check matrix and x_r runs through all permissible code symbols at time r. The state at the end of time interval $r+1$, s_{r+1} is calculated from

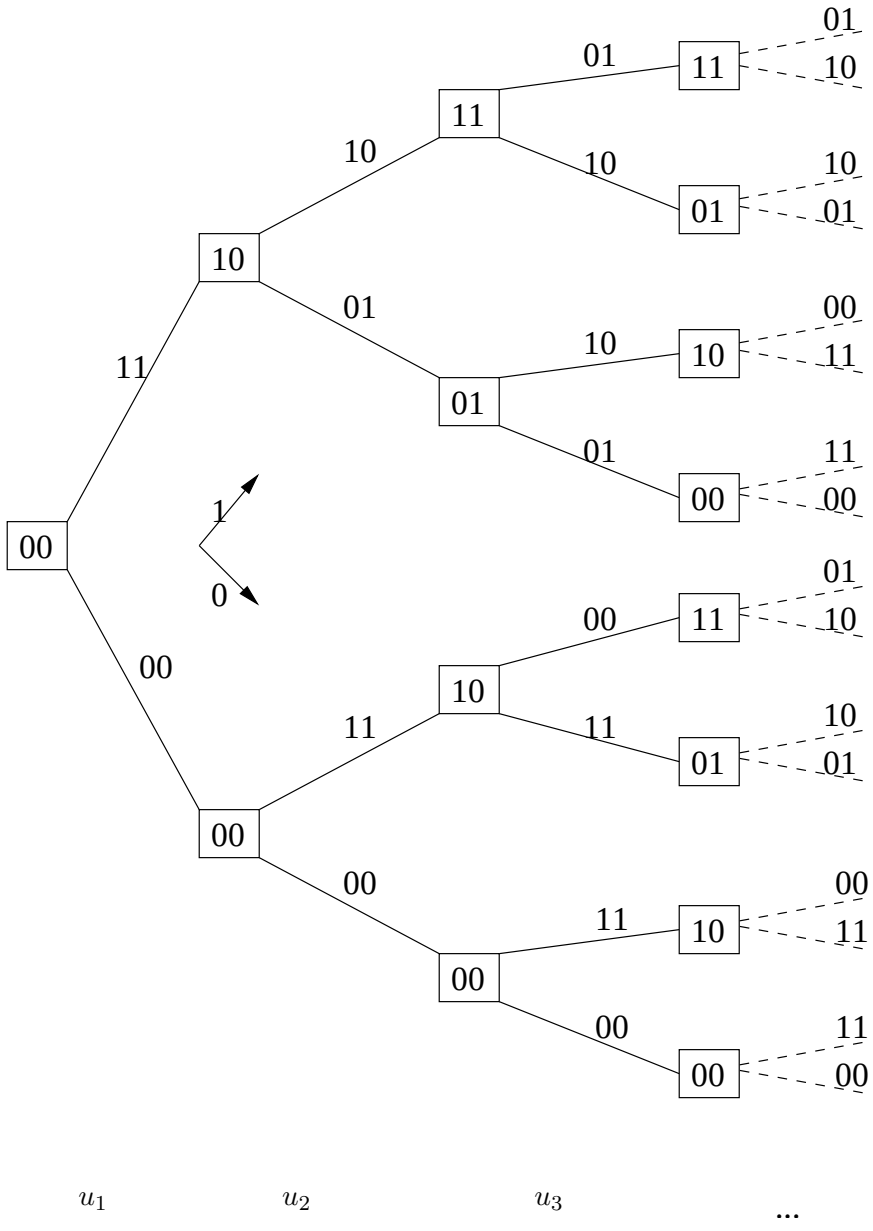

Figure 2.5. The tree representation of the convolutional code.

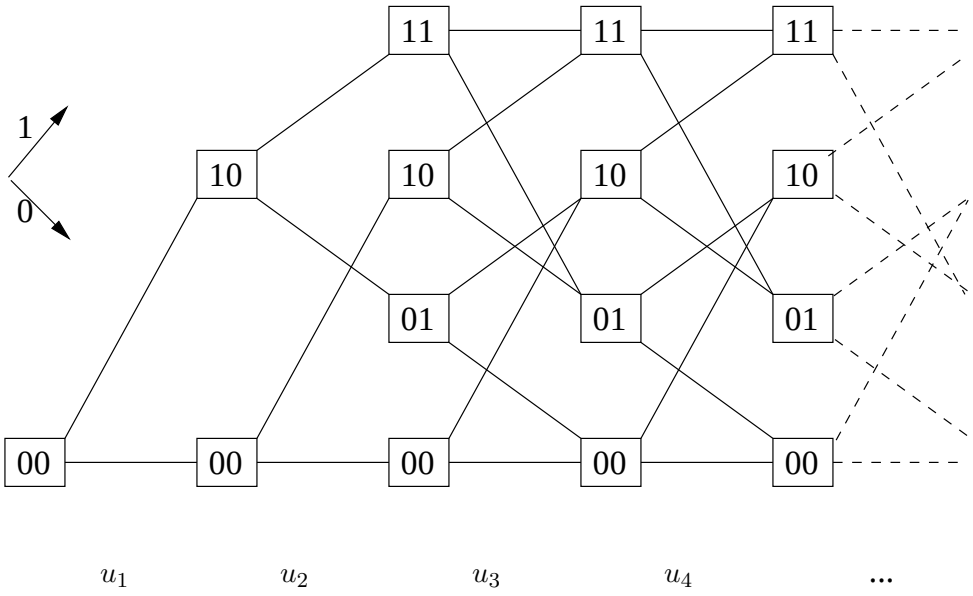

Figure 2.6. Trellis representation of the convolutional code.

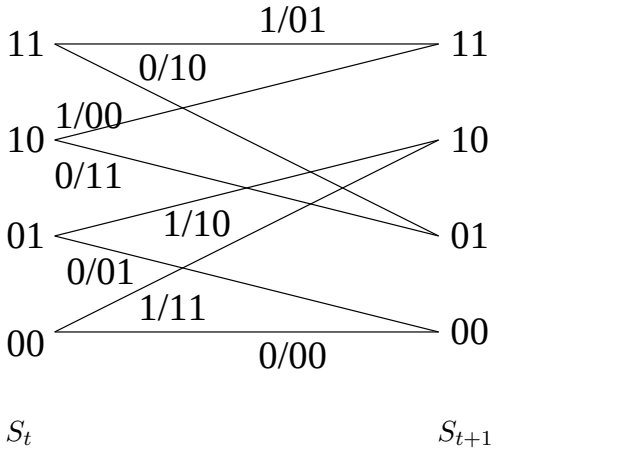

Figure 2.7. One trellis section of the convolutional code.

the preceding state s_r according to Equation 2.18. If the states s_r and s_{r+1} are connected, they are joined in the trellis diagram by a branch labeled with the output symbol x_r that causes the connection. If a certain value for x_r makes the final state unable to be the all 0 state, then this value for x_r is not permissible. Using this method, we can find the trellis of the extended Hamming (8,4,4) code defined by Equation 2.17 shown in Figure 2.8. In the figure, the horizontal branches denotes $x_r = 0$ where the up going or down going branches denotes $x_r = 1$. For example, if we choose $x_1 = 1, x_2 = 1, x_3 = 1$, then state $s_4 = 1000 + 1001 + 1101 = 1100$. At this point, $x_4 = 0$ is not permissible because no matter what we choose for x_5, x_6, x_7, x_8, the final state s_9 cannot be the all 0 state. As a result, we can only choose $x_4 = 0$. Then, if we choose $x_5 = 1, x_6 = 1, x_7 = 1, x_8 = 1$, the final state $s_9 = 0000$. As a result, 11111111 is a valid codeword, and there is a corresponding trellis path on the trellis diagram.

2.5.3 Tail-biting Trellis

As the code complexity goes up, the trellis representation also goes up more than linearly with the convolutional code's memory order or the block code's length. Some trellis sections, especially the mid trellis sections, are so complex that they make the hardware implementation a big problem. For the traditional trellis, there is only one node at the beginning and ending point. As a result, the trellis sections near the beginning point and ending point is comparatively simple as compared with the trellis sections at the mid point. In other words, the trellis is not well balanced. Can we make the trellis more balanced to decrease the overall complexity of the trellis? Calderbank [10] and other researchers gave the answer by using the tail-biting trellis. Figure 2.9 shows the structure of a tail-biting trellis (Notice that the trellis sections do not need to be identical). In the tail-biting trellis, there is no single starting point and ending point. Instead, any path that starts and ends at the same point represents a valid codeword.

For example, for the extended Hamming (8,4,4), there is a tail-biting trellis shown in Figure 2.10 [10]. In Figure 2.10, every trellis starts and ends at the same point is a valid codeword of the extended Hamming (8,4,4) code. For example, 11000110 is a valid codeword because it starts and ends at the same state. For the tail-biting trellis shown in Figure 2.10, the state-complexity profile is 2,4,4,4,2,4,4,4 while the state-complexity profile for the conventional trellis shown in Figure 2.8 is 1,2,4,8,4,8,4,2,1. The tail-biting trellis is simpler and easier to be realized than the conventional trellis.

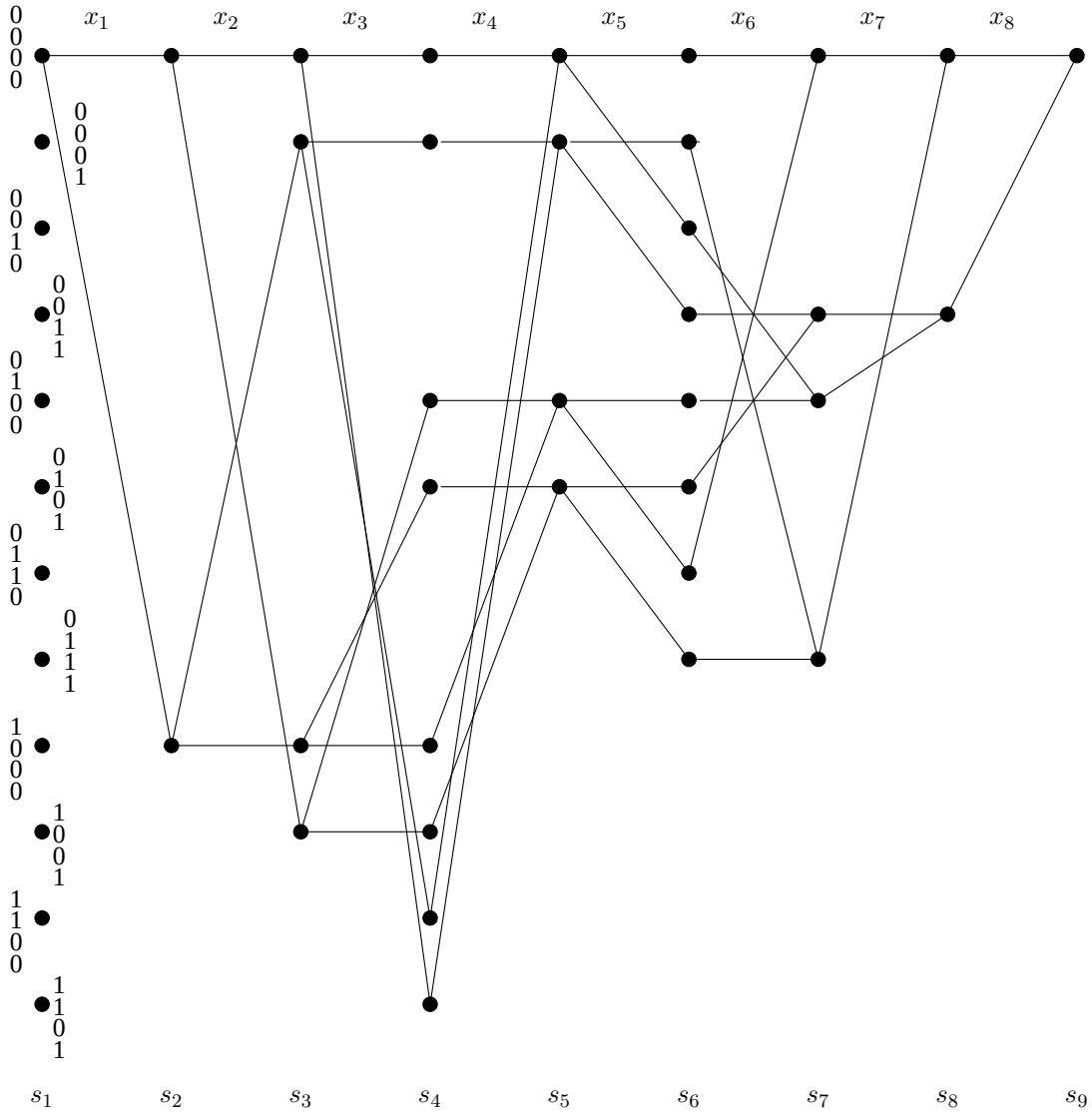

Figure 2.8. Trellis diagram of the extended Hamming (8,4,4) code.

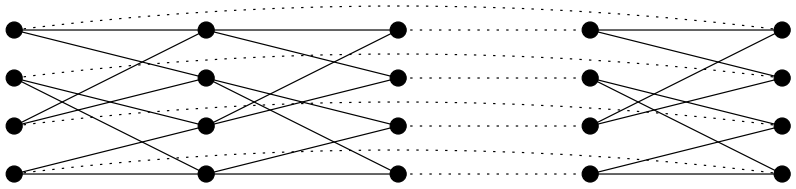

Figure 2.9. Structure of a tail-biting trellis.

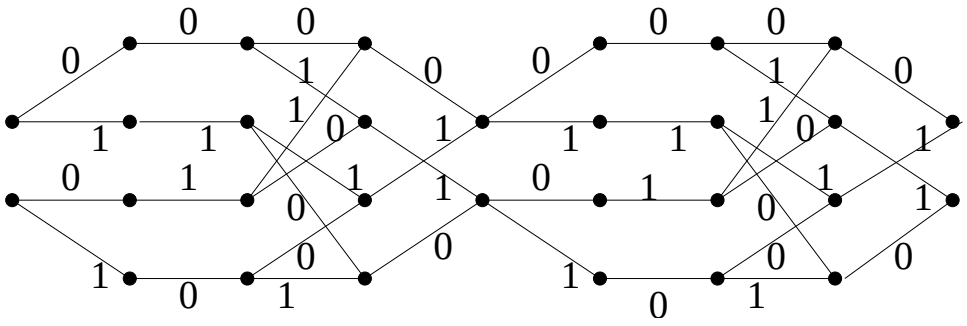

Figure 2.10. Tail-biting trellis for the extended Hamming (8,4,4) code.

If we combine adjacent trellis sections together and redraw Figure 2.10, we can get the figure shown in Figure 2.11 that is used for our extended Hamming (8,4,4) decoder [69]. In Figure 2.11, $u_1 = x_1, u_2 = x_2 + x_3, u_3 = x_5, u_4 = x_6 + x_7$ and the branches are labeled with the input/output of the encoder.

2.6 Noise Representation

Until now, we have not considered anything about noise. Of course, noise is important. Without knowing the noise model, we cannot know what is the probability of

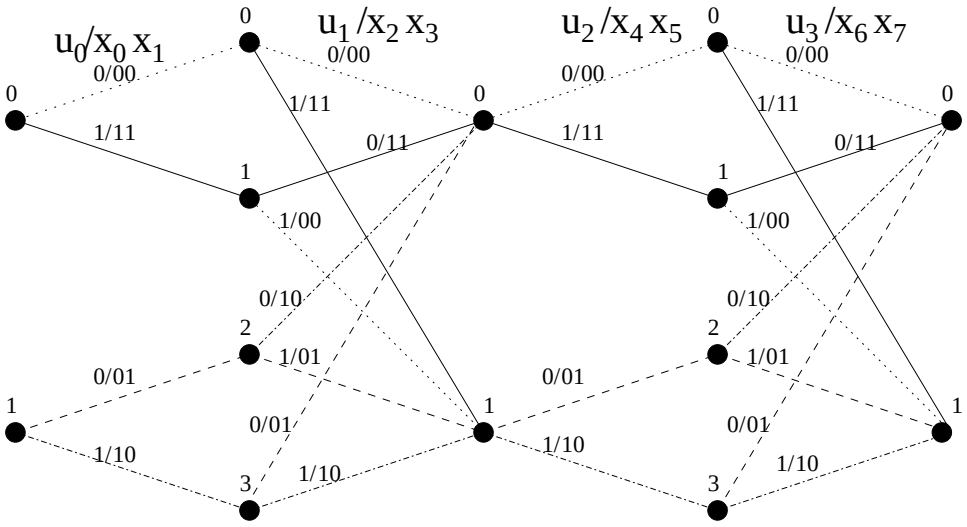

Figure 2.11. Compact view of the tail-biting trellis for the extended Hamming (8,4,4) code.

the received bit to be a '1' or '0'. In communication, there are a large number of noise models. However, the basic and most widely used noise model is the memory-less *additive white Gaussian noise, AWGN* model. This thesis concentrates only on the memory-less AWGN model. Figure 2.1 is a simplified communication model. In real applications, in order to send and receive information, a modulator and demodulator are used to make the transmitted signals on the channel continuous. As a result, a more complicated communication model is shown in Figure 2.12.

Now, using *BPSK (binary shift keying)* as an example, we show the relationship between the noise and the probabilities of a received bit to be a '0' or '1'. The following equation shows the two signals that BPSK uses. In the equation, E_c is the energy of each symbol and T_s is the duration of one symbol and f_0 is a multiple of $1/T_s$.

$$s_0(t) = \sqrt{\frac{2E_c}{T_s}} sin(2\pi f_0 t + \frac{\pi}{2}), 0 \leq t \leq T_s$$

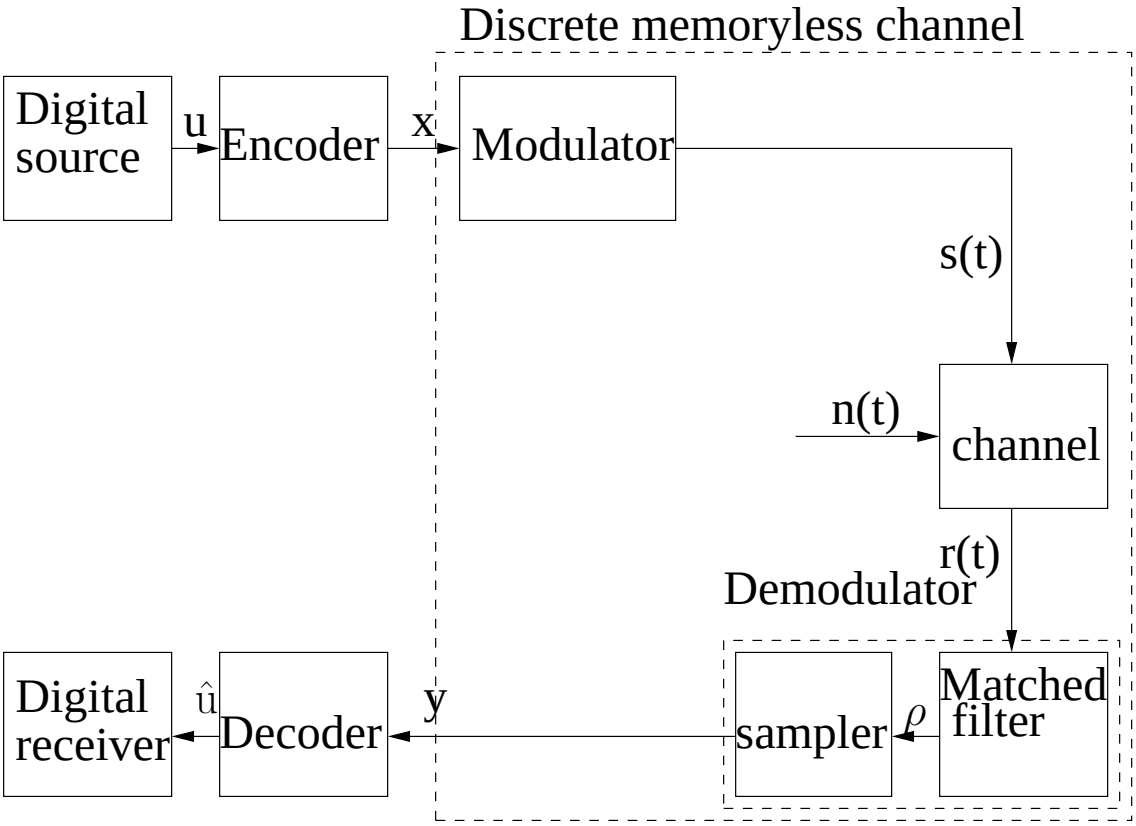

Figure 2.12. A more detailed model of a communication system.

$$s_1(t) = \sqrt{\frac{2E_c}{T_s}} sin(2\pi f_0 t - \frac{\pi}{2}), 0 \leq t \leq T_s \tag{2.19}$$

Since the noise is memory-less additive white Gaussian noise, the following equation is true.

$$r(t) = s(t) + n(t), 0 \leq t \leq T_s \tag{2.20}$$

In order to know the probability that the sender has sent out a '1', we need to know how much the received signal resembles the signal denoting a '1'. As a result, the following equation is used by the demodulator and this equation is implemented by a matched filter.

$$\rho = \int_0^{T_s} r(t) \sqrt{\frac{2}{T_s}} sin(2\pi f_0(t) + \frac{\pi}{2}) \, \mathrm{d}t \tag{2.21}$$

However, this equation is too complex to be used. Because the noise is white, each sample of the noise is independent to any other samples. As a result, the noise could be denoted by a real number with the probability density function defined by

$$f_n(x) = \frac{1}{\sqrt{2\pi\sigma_n^2}} e^{-\frac{x^2}{2\sigma_n^2}} \tag{2.22}$$

in which σ_n^2 is the variance of the zero-mean white Gaussian noise $n(t)$. This variance, σ_n^2, is related to the one-sided *power spectral density (PSD)* N_0 by

$$\sigma_n^2 = \frac{N_0}{2} \tag{2.23}$$

Since $s_1(t) = -s_0(t)$, we can normalize the noise power according to E_c and then use 1 to denote $s_1(t)$ and -1 to denote $s_0(t)$. Now, the equation between the transmitted bit, the received bit, and the noise can be represented by Equation 2.24. In the equation, x can only choose values 1 and -1 and n can be any real number with probability density function described by Equation 2.22.

$$y = x + n \tag{2.24}$$

Now, the conditional probability of knowing $x = 1$ based on the recevied value y is determined by Equation 2.25. This equation is easy to use compared with Equation 2.21.

$$\begin{aligned} p(x = 1|y) &= \frac{\frac{1}{\sqrt{2\pi\sigma_n^2}} e^{-\frac{(y-1)^2}{2\sigma_n^2}}}{\frac{1}{\sqrt{2\pi\sigma_n^2}} e^{-\frac{(y-1)^2}{2\sigma_n^2}} + \frac{1}{\sqrt{2\pi\sigma_n^2}} e^{-\frac{(y+1)^2}{2\sigma_n^2}}} \\ &= \frac{1}{1 + e^{-\frac{4y}{2\sigma_n^2}}} \end{aligned} \tag{2.25}$$

2.7 Soft Decision Versus Hard Decision

Traditionally, in the algebraic decoding of block codes, we determine first whether the received information is a '1' or '0'. Then, we use the parity-check equations to find out what the information bits should be. For the decoding of convolutional codes, it is also determined whether the received bit is a '1' or '0' first before finding a trellis path with minimum distance to the determined value. The first step of deciding whether the information is a '1' or '0' is called a *hard decision*. However, in the hard decision, we lose some information. For example, no matter how close the information bit is near to a '1', it is considered to be a '1' as long as it has more distance to a '0'. Even if in situation s_1, the information bit is much closer to a '1' than in situation s_2, the two situations are treated as the same and part of the information is lost. However, if we use probability to mean how close the information is near to a '1' or '0', then the probability of being a '1' in situation s_1 is much higher than in situation s_2. As a result, no information is lost. This kind of decision is called a *soft decision*. Soft decision is superior to hard decision in error rate, but is requires a more complex decoder.

Let us use the trellis of the extended Hamming (8,4) code shown in Figure 2.8 as an example. Suppose that the probabilities of the received bits are shown in Table 2.1. If hard decision is used, then in the first step it is decided that sequence 11100000 is received, then by either using algebraic equations or finding a trellis path with minimum distance to the determined value, the decoded sequence is decided to be 11110000 and the information bit is decided to be 1000. However, using soft decision, the decoded sequence is decided to be 00000000 and the information bit is decided to be 0000. From the example, we can see that the probability of x_1, x_2, and x_3 being '0' or '1' are nearly equal. However, the probability of x_4 being '0' is much larger than its probability of being '1'. Using soft decision, x_4 helps us find a better result.

Table 2.1. Example of received bits of the extended Hamming (8,4) code.

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
Probability of being '0'	0.49	0.49	0.49	0.99	0.99	0.99	0.99	0.99
Probability of being '1'	0.51	0.51	0.51	0.01	0.01	0.01	0.01	0.01

2.8 Decision Rule

So far, we have described the code structure and the channel model (noise model). We have not, however, described the decoder. Of course, the decoder is needed to decide what the source information is. A *decision rule* is a rule that the decoder uses to determine the source information. Of course, we would like to have a good decision rule that can make a good decision based on what has been received. Two important decision rules are *maximum-a-posterior (MAP)* decision rule and *maximum-likelihood (ML)* decision rule and we describe them in the following subsections.

2.8.1 MAP Decision and ML Decision

Using a soft decision rule, we would like to choose the decoder output to be the one with maximum conditional probability when we receive a sequence y as shown in Figure 2.12. Thus, the maximum-a-posterior (MAP) decision rule is introduced. Suppose that all the valid information sequences u construct the information space U and all the received sequences y construct the space Y, then a MAP decoder can be defined as

$$\tilde{u}_{MAP}(y) = max\left[P(u|y)\right] \tag{2.26}$$

in which the function $max[P(u|y)]$ returns the particular u that maximizes the probability $P(u|y)$. This decision rule is called the maximum-a-posterior (MAP) decision rule because it maximizes the a posterior probability $p(u|y)$ for a certain y. Using the MAP decision rule with trellis coding, Bahl, Cocke, Jelinek, and Raviv [4] presented the BCJR algorithm (also called the *forward-backward algorithm*) that is widely used.

Because $P_Y(y)$ is the statistical probability of receiving sequence y that is irrelevant to u , Equation 2.26 can be written as the following equation.

$$\tilde{u}_{MAP}(y) = max\left[P(u|y)\right] = max\left[\frac{P(u,y)}{P_Y(y)}\right] = max\left[P(u,y)\right] \tag{2.27}$$

$P_U(u)$ is the probability distribution of sequence u being sent in the set U. In most cases, it is a constant and uniformly distributed. As a result, the equation can be further simplified as follows.

$$\tilde{u}_{ML}(y) = max\left[P(u,y)\right] = max\left[P(y|u)P_U(u)\right] = max\left[P(y|u)\right] \tag{2.28}$$

Now, instead of maximizing the conditional probability of having sent out u when we know we have received y, we maximize the conditional probability of sending out u and

receiving y. This probability is much more directly related to the channel characteristics. The decision rule using Equation 2.28 is called the maximum-likelihood (ML) decision rule. As a result, in general, the ML decision rule can be more easily realized. The famous Viterbi algorithm [66], [18] uses the ML decision rule. In cases in which $P_U(u)$ is uniformly distributed, the ML decision rule and the MAP decision rule are equivalent.

Also, from Figure 2.12, we know there is a one to one projection between the original information sequence and the encoded output sequence x. As a result, Equation 2.29 and Equation 2.30 can be derived. For a memoryless channel, Equation 2.31 and Equation 2.32 can be derived.

$$\tilde{u}_{MAP}(y) = max\left[P(u|y)\right] = max\left[P(x|y)\right] \tag{2.29}$$

$$\tilde{u}_{ML}(y) = max\left[P(y|u)\right] = max\left[P(y|x)\right] \tag{2.30}$$

$$\tilde{u}_{MAP}(y) = max\left[\prod_{i=1}^{n} P(x_i|y_i))\right] \tag{2.31}$$

$$\tilde{u}_{ML}(y) = max\left[\prod_{i=1}^{n} P(y_i|x_i)\right] \tag{2.32}$$

2.8.2 Block-Wise Decision and Bit-Wise Decision

Let us use u to mean the information sequence, u_k to mean a certain information bit and y to mean the received sequence as stated in Subsection 2.2.1 and let U to mean the source codeword space composed by all the source codewords. Then, if Equation 2.29 or 2.30 is used to determine the information bits, it is called a *block-wise decision*, also known as sequence estimation because it estimates the probability of the received sequence to be a sequence in the trellis. The Viterbi algorithm uses the block-wise ML decision. For example, using the probabilities shown in Table 2.1 and the trellis shown in Figure 2.8, sequence 00000000 has the largest probability $0.49*0.49*0.49*0.99*0.99*0.99*0.99*0.99$ in the trellis so that it is chosen to be the decoded result.

Instead, if Equation 2.33 or Equation 2.34 is used to determine each bit of the information bits, it is called a *bit-wise decision* because they sum over the probabilities of

all the codewords that have the certain information bit $u_k = 0$ and the probabilities of all the codewords that have the certain information bit $u_k = 1$ to get the probability of this certain bit u_k to be a '0' or '1' and then make the decision based on the probabilities of this certain information bit. For a memoryless channel, Equation 2.35 and Equation 2.36 can be derived. For example, if we need to determine x_5 that is also information bit u_3 shown in Figure 2.11, we need to determine the probabilities of all valid trellis paths. If the sum of the probabilities of all the valid trellis paths that have x_5 equal '0' is larger than the sum of the probabilities of all the valid trellis paths that have x_5 equal '1', then x_5 is decided to be '0', otherwise it is decided to be '1'.

$$\tilde{u}_{kMAP}(y) = max_{u_k}\left[P(u_k|y)\right] = max_{u_k}\left[\sum_{u\in U,\tilde{u_k}=u_k} P(u|y)\right] \tag{2.33}$$

$$\tilde{u}_{kML}(y) = max_{u_k}\left[P(y|u_k)\right] = max_{u_k}\left[\sum_{u\in U,\tilde{u_k}=u_k} P(y|u)\right] \tag{2.34}$$

$$\tilde{u}_{kMAP}(y) = max_{u_k}\left[\sum_{u\in U,\tilde{u_k}=u_k}\prod_{i=1}^{n} P(x_i|y_i)\right] \tag{2.35}$$

$$\tilde{u}_{kML}(y) = max_{u_k}\left[\sum_{u\in U,\tilde{u_k}=u_k}\prod_{i=1}^{n} P(y_i|x_i)\right] \tag{2.36}$$

Using bit-wise decision, the information bit is decided only by the channel information and context without the need of forcing the received sequence to be a valid sequence. As a result, bit-wise decision is superior to block-wise decision. However, from the previous description, bit-wise decision is much more difficult to be implemented than block decision.

2.8.3 Bit-Wise MAP Decision

The previous description indicates that the bit-wise MAP decision rule is the optimum. This subsection discusses the method of using bit-wise MAP decision on trellis coding.

Let us define s_r to mean the state at time r and s_{r+1} to mean the state at time $r+1$. In order to find $max_{u_k}\left[P(u_k|y)\right]$, we need to first find the probability shown in Equation 2.37.

$$P(s_r = i, s_{r+1} = j|y) \tag{2.37}$$

Since Equation 2.38 is true and the maximization of $max_{u_k}\,[P(u_k|y)]$ is independent of $P(y)$, we can evaluate the joint probabilities instead of the conditional probabilities.

$$P(s_r = i, s_{r+1} = j|y) = \frac{P(s_r = i, s_{r+1} = j, y)}{P(y)} \tag{2.38}$$

Before we proceed, let us define $\alpha_r(j) = P(s_r = j, \tilde{y})$ to be the joint probability of the partial sequence $\tilde{y} = (y_{-l}, \cdots, y_{r-1})$ up to and including time epoch $r-1$ and state $s_r = j$, $\beta_r(j) = P((y_r, \cdots, y_l)|s_r = j)$ to be the conditional probability of the remainder of the received sequence y given that the state at time r is j and $\gamma_r(i,j)$ to be the joint conditional probability of y_r and the state at time $r+1$ equals j, given that the state at time r is i. Then, using Bayes' rule, the following equations can be derived.

$$\begin{aligned} P(s_r = i, s_{r+1} = j, y) &= P(s_r = i, s_{r+1} = j, (y_{-l}, \cdots, y_{r-1}), y_r, (y_{r+1}, \cdots, y_l)) \\ &= P(s_r = i, (y_{-l}, \cdots, y_{r-1}))P(s_{r+1} = j, y_r|s_r = i) \\ &\quad \cdot P((y_{r+1}, \cdots, y_l)|s_{r+1} = j) \\ &= \alpha_r(i)\gamma_r(i,j)\beta_{r+1}(j) \end{aligned} \tag{2.39}$$

$$\begin{aligned} \alpha_r(j) &= \sum_{states\ i} P(s_{r-1} = i, s_r = j, \tilde{y}) \\ &= \sum_{states\ i} P(s_{r-1} = i, (y_{-l}, \cdots, y_{r-1}))P(s_r = j, y_r|s_{r-1} = i) \\ &= \sum_{states\ i} \alpha_{r-1}(i)\gamma_{r-1}(i,j) \end{aligned} \tag{2.40}$$

$$\begin{aligned} \beta_r(j) &= \sum_{states\ i} P(s_{r+1} = i, (y_r, \cdots, y_l)|s_r = j) \\ &= \sum_{states\ i} P(s_{r+1} = i, y_r|s_r = j)P((y_{r+1}, \cdots, y_l)|s_{r+1} = i) \\ &= \sum_{states\ i} \beta_{r+1}(i)\gamma_r(j,i) \end{aligned} \tag{2.41}$$

For trellis codes starting in the zero state at time $r = -l$ and end in the zero state at time $r = l+1$, the boundary condition is shown in Equation 2.42.

$$\alpha_{-l}(0) = 1,\ \alpha_{-l}(j) = 0(\ j \neq 0)$$
$$\beta_{l+1}(0) = 1,\ \beta_{l+1}(j) = 0(\ j \neq 0) \tag{2.42}$$

Using the boundary condition and Equation 2.40, we can process forward along the trellis to get all the α values. Using the boundary condition and Equation 2.41, we can process backward along the trellis to get all the β values. Then, by using Equation 2.39, we can find the answer for the joint probabilities $P(s_r = i, s_{r+1} = j, y)$. As a result, this algorithm is often called the forward-backward algorithm.

If we get the probability by summing over the probabilities shown in Equation 2.43 instead of the probabilities shown in Equation 2.39, we get the *extrinsic information* without using the channel information for the current bit. The extrinsic information provides the context information for the current bit and is often used in iterative decoding.

$$\begin{aligned} P(s_r = i, (y_{-l}, \cdots, y_{r-1}, y_{r+1}, \cdots, y_l)|s_{r+1} = j) &= P(s_r = i, (y_{-l}, \cdots, y_{r-1})) \\ & \quad P((y_{r+1}, \cdots, y_l)|s_{r+1} = j) \\ &= \alpha_r(i)\beta_{r+1}(j) \end{aligned} \tag{2.43}$$

2.9 Iterative Decoding

As the code complexity increases, the complexity of using the MAP decision rule increases more than linearly, making it difficult to be implemented. A better idea is to not decode the information all at once. Instead, we decode only using part of the related information that has been received. Then, step by step, we use other parts of the related information to generate a better result until all the related information is used and the decode result has stabilized to a nearly optimum result. *Iterative decoding* is the decoding strategy using this technique. For example, if we have received a larger sequence y in which the information bits are related, we can use part of y to decode first and get the decoded result. Then, we can gradually add the other parts of y into the decoding process and finally find a near optimum result. In doing iterative decoding, for every component decoder, we need to use both the channel information and the extrinsic information generated by the previous step so that the context information provided by other component decoders can be used. As a result, usually bit-wise MAP decision is used for each component code so that it can give the extrinsic information. Iterative

decoding is used in a large number of decoders such as Turbo decoders [9], [8], [7], [6] LDPC decoders [19], [47], [41], [57], [46] and product decoders [25], [49], [72], [70] that we discuss in the next section.

2.10 Product Codes

It is long known that even for the simple even parity-check, using a two-dimensional code can gain a much better protection than using a one-dimensional code. Figure 2.13 shows the structure of the two-dimensional even parity-check code. By using two-dimensional even parity-check, a relationship is created in the $k_1 * k_2$ bits and the redundancy provided by the even parity-check bits is more useful and more protection is gained compared with the one-dimensional even parity-check.

By expanding this idea to general codes, product codes of multiple dimensions can be built. Figure 2.14 shows the structure of a two-dimensional product code with component codes C_1 and C_2. In theory, the component codes can be either block codes or convolutional codes. Usually, the component codes are linear block codes. Also, if the parity on parity is not provided, it is called a "punctured version." If the parity on parity is provided, it is called a "full version." Since the product code creates a relationship between $n_1 * n_2$ bits, it provides better performance than its component code. Simulation has shown that the product code has an amazing performance despite its simple structure.

The decoder for a product code is realized by iterative decoding. From Figure 2.14, we know that the row code and column code intersect each other and share the same

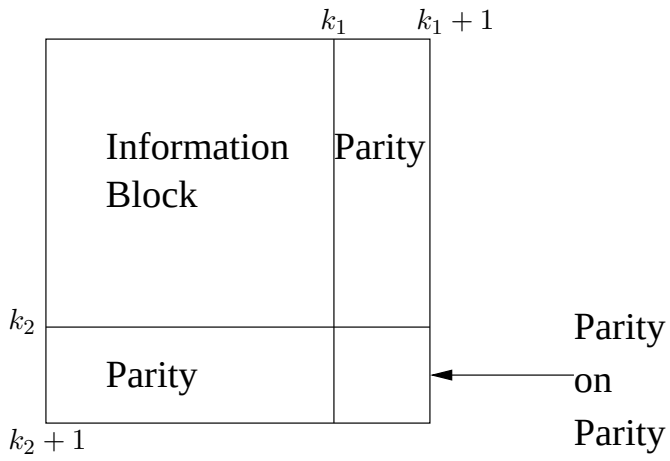

Figure 2.13. Two-dimensional even parity-check.

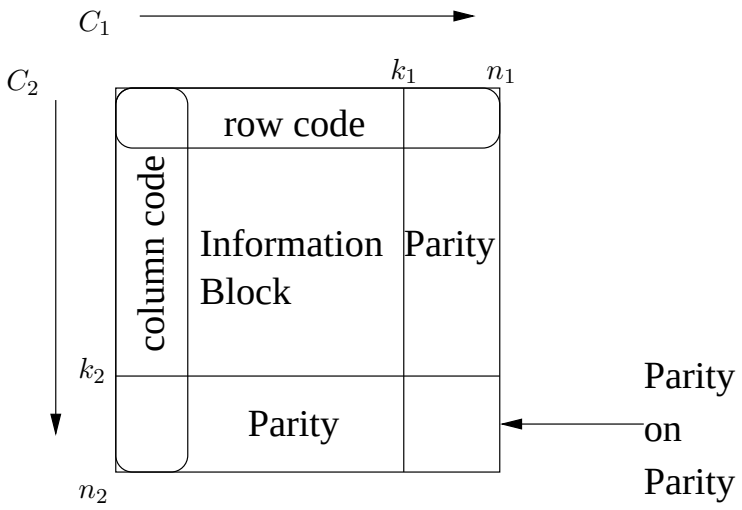

Figure 2.14. Two-dimensional product code.

information bit. Thus, they have a relationship. As a result, there is extrinsic information exchanges between the row code and the column code. Actually the relationship and the decoding of the information bit can be realized by an *equal gate* shown in Figure 2.15. For an equal gate that has connections with n nodes $1, 2, \cdots, n$, the function of the equal gate is expressed by Equation 2.44 in which $p_i(0)$ means the probability of node i to be 0 and $p_i(1)$ means the probability of node i to be 1 and $p'_i(0)$ and $p'_i(1)$ represent the new probability of node i. For the equal gate shown in Figure 2.15, the channel information γ is only provided as input and no output for it is needed. As a result, the function of the equal gate shown in Figure 2.15 can be expressed by Equation 2.45 to Equation 2.47. In most cases, the a prior probability p_u has a unity distribution $p_u(0) = p_u(1)$. As a result, Equation 2.45 and Equation 2.46 can be simplified to not include p_u.

$$\begin{aligned} p'_j(0) &= \frac{\sum_{i=1,i\neq j}^{n} p_i(0)}{\sum_{i=1,i\neq j}^{n} p_i(0) + \sum_{i=1,i\neq j}^{n} p_i(1)} \\ p'_j(1) &= \frac{\sum_{i=1,i\neq j}^{n} p_i(1)}{\sum_{i=1,i\neq j}^{n} p_i(0) + \sum_{i=1,i\neq j}^{n} p_i(1)} \quad (j = 1, \cdots, n) \end{aligned} \tag{2.44}$$

$$p'_r(0) = \frac{p_u(0)p_\gamma(0)p_c(0)}{p_u(0)p_\gamma(0)p_c(0) + p_u(1)p_\gamma(1)p_c(1)}$$

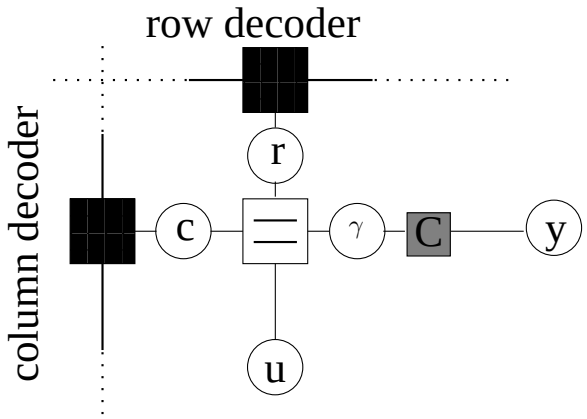

Figure 2.15. The equal gate connecting a row decoder and a column decoder.

$$p'_r(1) = \frac{p_u(1)p_\gamma(1)p_c(1)}{p_u(0)p_\gamma(0)p_c(0) + p_u(1)p_\gamma(1)p_c(1)} \tag{2.45}$$

$$\begin{aligned} p'_c(0) &= \frac{p_u(0)p_\gamma(0)p_r(0)}{p_u(0)p_\gamma(0)p_r(0) + p_u(1)p_\gamma(1)p_r(1)} \\ p'_c(1) &= \frac{p_u(1)p_\gamma(1)p_r(1)}{p_u(0)p_\gamma(0)p_r(0) + p_u(1)p_\gamma(1)p_r(1)} \end{aligned} \tag{2.46}$$

$$\begin{aligned} p'_u(0) &= \frac{p_\gamma(0)p_r(0)p_c(0)}{p_\gamma(0)p_r(0)p_c(0) + p_\gamma(1)p_r(1)p_c(1)} \\ p'_u(1) &= \frac{p_\gamma(1)p_r(1)p_c(1)}{p_\gamma(0)p_r(0)p_c(0) + p_\gamma(1)p_r(1)p_c(1)} \end{aligned} \tag{2.47}$$

As a result, the row decoder and column decoder exchange extrinsic information. For every iteration, the row and column decoder use not only the a prior probability p_u and the channel information, but also the extrinsic information from another component decoder (Notice that for the simple extended Hamming (8,4) decoder, only the a priori probability and the channel information are used). As a result, for every iteration, the component decoder uses not only the context information provided by the the current component decoder, but also the context information provided by other component decoders that is generated in the previous step. Thus, more redundancy is used by the component decoder and the result of the component decoder is improved gradually until it stabilizes and reaches a near optimum result. In the decoding, the channel information and the extrinsic information provided by the row decoder and column decoder are both used to generate a good result.

CHAPTER 3

FACTOR GRAPH SIMULATION

As stated in Chapter 1, researchers have observed that a number of important algorithms in error-control coding can be interpreted as operations of the *sum-product algorithm* on *probability propagation networks* that is a kind of *factor graph* [17] [35] [2]. This chapter discusses factor graphs, how to apply the sum-product algorithm to factor graphs, and how to use a factor graph to model a coding system. Finally, this chapter discusses how to do high-level simulation of an error control decoder from its factor graph model and how to automate the high-level simulation.

3.1 Factor Graph

Factor graphs can be used to model coding systems. This section introduces the concept of a factor graph. Then, it discusses how to use factor graphs to construct the behavioral model and probabilistic model of a coding system.

3.1.1 Definition of Factor Graph

According to the definition in [35], a factor graph is a bipartite graph that expresses the structure of the factorization shown in Equation 3.1 in which a "global" function of many variables is factored into a product of "local" functions. In Equation 3.1, J is a discrete index set, X_j is a subset of $x_1, \cdots, x_n$ and $f_j(X_j)$ is a function having the elements of X_j as arguments. A factor graph has a *variable node* for each variable x_i, a function node for each local function f_j and an edge connecting variable node x_i to function node f_j if and only if x_i is an argument of f_j. For example, the function g defined by $g(x, y) = xy - x$ can be factored into $g(x, y) = f_1(x)f_2(y)$ where $f_1(x) = x$ and $f_2(y) = y - 1$. The factor graph depicting this factorization is shown in Figure 3.1.

$$g(x_1, \cdots x_n) = \prod_{j \in J} f_j(X_j) \tag{3.1}$$

Figure 3.1. Factor graph example 1.

In Figure 3.1, the circles represent the variables and the squares represent the functions. A line, or edge, is drawn between a circle (variable) and a square (function) if the variable for the circle is an argument of the function for the square. Thus, the circle, edge, and square on the left represent $f_1(x)$ and the circle, edge, and square on the right represent $f_2(y)$. The "global" function is the product of the local functions, i.e., $g(x, y) = f_1(x)f_2(y)$.

As another example, consider the factor graph in Figure 3.2. The three circles represent the 3 variables x, y, z and the three squares represent the three functions f_1, f_2, f_3. The edges connecting the circles and squares tell us that this graph represents a function g that factors as $g(x, y, z) = f_1(x, y)f_2(y, z)f_3(x, z)$.

3.1.2 Configuration and Behavioral Modeling

In many applications, the function is defined by a configuration space. For example, for a binary linear (n, k) block code, there is a code space that is constructed by 2^k codewords. If a length n codeword is a codeword in the code space, it is a valid codeword, otherwise it is not a valid codeword. In this way, we can define a function based on the code space. The function has the domain of length n codewords. If a codeword is a valid codeword, the function has value 1, which means true, otherwise the function has value 0, which means false. In general, let $x_1, x_2, \cdots, x_n$ be a collection of variables with

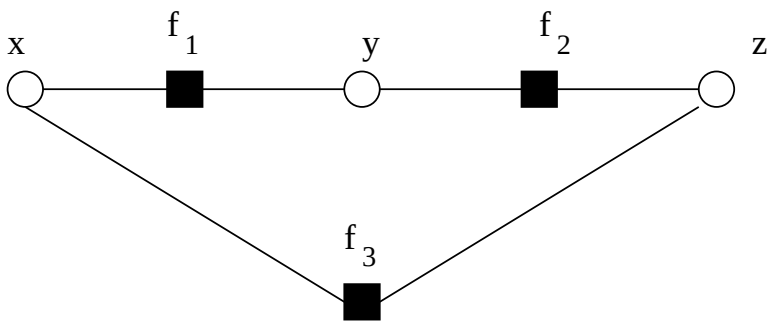

Figure 3.2. Factor graph example 2.

configuration space $S = A_1 * A_2 * \cdots * A_n$. Let B be a subset of S that means valid *behavior*. The elements of B are the *valid configurations*. Let us define a function g with parameters $x_1, x_2, \cdots, x_n$. If the parameter's value is an element of B, g has value 1, indicating its a valid configuration. Otherwise g has value 0. Because g indicates whether the configuration is valid or not, we call g an *indicator function*. Because the indicator function is defined by the behavior B, this approach is known as behavioral modeling [68].

For a "global" indicator function, it may be factored into the product of "local" indicator functions. Each "local" function has its variables and configuration space. If all the "local" indicator functions have value 1, then the "global" indicator function has value 1. Otherwise the "global" indicator function has value 0. As a result, we can use a factor graph to show the factorization of the "global" indicator function. For a linear (n, k) block code, it is defined by a $(n-k, n)$ parity-check matrix in which $n-k$ parity-check equations are defined. For a length n codeword, if all the $n-k$ parity-check equations are satisfied, then the codeword is a valid codeword in the code space. Of course, the "global" indicator function can be factored into the product of $n-k$ "local" functions whose configuration spaces are defined by the $n-k$ parity-check equations. For example, if we use C to show the code space of the extended Hamming (8,4) code, which is defined by the parity-check matrix shown in Equation 2.17, the factorization of the "global" indicator function of the extended Hamming (8,4) code is shown in Equation 3.2 in which "+" means XOR operation in the binary field. The corresponding factor graph is shown in Figure 3.3 in which a square with a "+" sign is used to represent the parity-checks. A factor graph obtained in this way is often called a *Tanner graph* [63].

$$\begin{aligned} g(x_1, x_2, \cdots x_8) &= [(x_1, x_2, \cdots x_8) \in C] \\ &= [x_1 + x_2 + x_3 + x_4 = 0]\,[x_3 + x_4 + x_6 + x_7 = 0] \\ &\quad [x_5 + x_6 + x_7 + x_8 = 0]\,[x_2 + x_3 + x_7 + x_8 = 0] \end{aligned} \tag{3.2}$$

3.1.3 Probabilistic Modeling

Since conditional and unconditional independence of random variables is expressed in terms of a factorization of their joint probability mass or density function, factor graphs for probability distributions can be used in many situations. Because the "global"

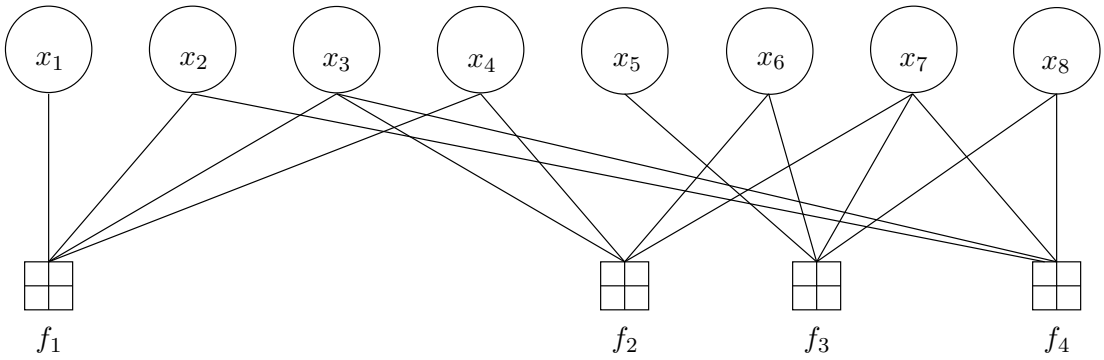

Figure 3.3. The Tanner graph of the extended Hamming (8,4) code shown in Equation 2.17.

function defines the joint probability mass or density function, this approach is known as *probabilistic modeling*. Also, the values that are passed between nodes are always probabilities, this kind of factor graph is known as *probability propagation network*. These are commonly used in decoding field. In decoding, MAP decision and ML decision can be used to decide the result. Probabilities are used in making the decision. Thus, instead of making a decision of whether the codeword is a valid codeword or not, we need to decide the probability of each of the valid codewords based on the received information and then make a decision according to the MAP or ML decision rule. If the block-wise MAP decision rule is used, then the probability of each of the valid codewords is based on the received information as shown in Equation 3.3 in which $f(x|y)$ denotes the conditional probability of knowing x has been sent based on the received value y. If the channel is memoryless, then Equation 3.4 can be derived. The corresponding factor graph is shown in Figure 3.4.

$$\begin{aligned} g(x_1, x_2, \cdots x_8) &= [x_1 + x_2 + x_3 + x_4 = 0]\,[x_3 + x_4 + x_6 + x_7 = 0] \\ &\quad [x_5 + x_6 + x_7 + x_8 = 0]\,[x_2 + x_3 + x_7 + x_8 = 0]\, f(x|y) \end{aligned} \tag{3.3}$$

$$\begin{aligned} g(x_1, x_2, \cdots x_8) &= [x_1 + x_2 + x_3 + x_4 = 0]\,[x_3 + x_4 + x_6 + x_7 = 0] \\ &\quad [x_5 + x_6 + x_7 + x_8 = 0]\,[x_2 + x_3 + x_7 + x_8 = 0] \prod_{i=1}^{8} f(x_i|y_i) \end{aligned} \tag{3.4}$$

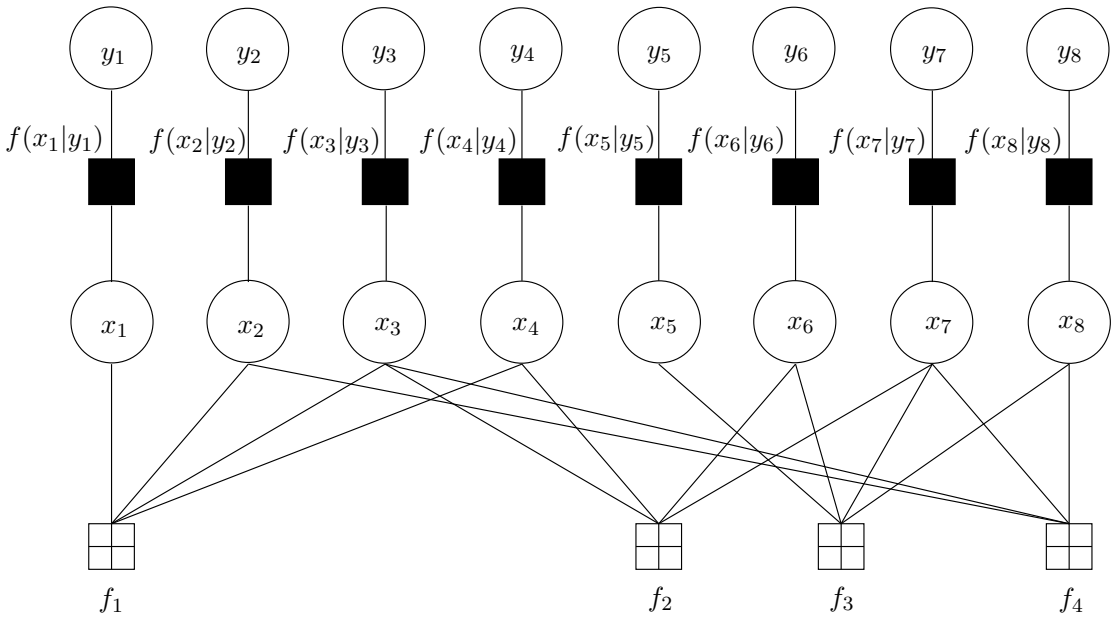

Figure 3.4. Factor graph for the conditional probability density function of the extended Hamming (8,4) decoder based on Equation 2.17.

3.2 The Sum Product Algorithm

In this section, we discuss the sum product algorithm. We begin with the introduction of marginal function and then introduce the sum-product operation. Then we discuss the sum product update rules and use some examples of the sum product algorithm applications to explain the sum product algorithm.

3.2.1 Marginal Function

In many situations (for example, when the "global" function $g(x_1, \cdots, x_n)$ represents a joint probability mass function), we are interested in computing the *marginal function* $g_i(x_i)$ as shown in Equation 3.5 in which $\sim \{x_i\}$ denotes the sum over all values of $x_1, ..., x_N$ except x_i. For the bit-wise MAP decision shown in Equation 2.33, it sums all the probabilities of the valid codewords corresponding to a certain information bit u_i based on the received information y. If we write the equation in a different form as shown in Equation 3.6 in which the function $[u \in U]$ is an indicator function indicating whether the information sequence u is a valid information sequence or not, then it is clear that the operations beside the *max* operation is implemented by a marginal function. If the "global" function can be factored into the product of "local" functions as shown in

Equation 3.1, then Equation 3.7 can be derived from Equation 3.5. In Equation 3.7, the first operation is generating the product and the next operation is summing the product terms.

$$g_i(x_i) = \sum_{\sim\{x_i\}} (g(x_1, ..., x_N)) \tag{3.5}$$

$$\tilde{u}_{iMAP}(y) = max_{u_i} \left[\sum_{\sim\{u_i\}} [u \in U] P(u|y) \right] \tag{3.6}$$

$$g_i(x_i) = \sum_{\sim\{x_i\}} (\prod_{j \in J} f_j(X_j)) \tag{3.7}$$

Also, by ordering the operations, we can do the sum-product operation recursively. The factor graph for an even parity-check code with length 4 shown in Figure 3.5 is used as an example. For this example, the "global" indicator function $g = (x_1 + x_2 + x_3 + x_4 = 0)$ is a function that indicates whether $x_1 + x_2 + x_3 + x_4 = 0$ is true.[1] This "global" indicator function can be factored into the product of two "local" indicator functions, $f_1 = (x_1 + x_2 + s = 0)$ that indicates whether $x_1 + x_2 + s = 0$ is true and $f_2 = (s + x_3 + x_4 = 0)$ that indicates whether $s + x_3 + x_4 = 0$ is true. If both f_1 and f_2 are true, then g is true, otherwise g is false. Thus, g is factored into the product of f_1 and f_2. Note that s is a new internal state variable that communicates information between the two functions. For example, if $x_1 = x_2 = 1$ then this implies that s must be 0 to make f_1 true. The fact that s must be 0 means that x_3 and x_4 must be equal to make f_2 true. In other words, g is true if and only if a consistent value for s can be found that makes both f_1 and f_2 true.

To determine the probability of x_3 shown in Figure 3.5 to be '0' and to be '1' (i.e., $p(x_3 = 0)$, $p(x_3 = 1)$) after decoding, probabilistic modeling is used. Using bit-wise MAP decision rule, the decoded result should be derived using Equation 3.8 that is a marginal function. In Equation 3.8, the functions $f(x_i|y_i)(i = 1, \ldots, 4)$ are channel functions expressed by Equation 2.25.

[1] 1 is used to mean true and 0 is used to mean false.

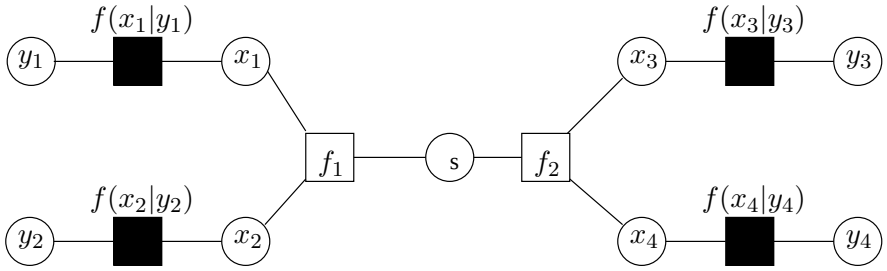

Figure 3.5. Factor graph for the even parity-check code with length 4.

$$g_3(x_3) = \sum_{\sim\{x_3\}} \left(g(x_1 + x_2 + x_3 + x_4 = 0) \prod_{i=1}^{4} f(x_i|y_i) \right) \tag{3.8}$$

By factoring the "global" indicator function into the product of "local" indicator functions and then ordering the operations, Equation 3.9 can be derived.

$$\begin{aligned} g_3(x_3) \quad = \quad & \left(\sum_{x_4,s} f_2(x_3 + x_4 + s = 0) f(x_4|y_4) \left(\sum_{x_1,x_2} f_1(x_1 + x_2 + s = 0) f_(x_1|y_1) f(x_2|y_2) \right) \right) \\ & f(x_3|y_3) \end{aligned} \tag{3.9}$$

When doing the computation recursively expressed by Equation 3.9, the process is just like calculating the result on an expression tree. Actually, we can draw the factor graph as a tree structure as shown in Figure 3.6. In Figure 3.6, the calculation begins from the leaf node. When the values y_1, y_2, y_3, and y_4 are received, these values are sent to the channel functions respectively. When the channel functions receive the messages containing the $y_i (i = 1, \ldots, 4)$, it can calculate the conditional probabilities according to Equation 2.25[2] and then send these probabilities as messages to the corresponding variable nodes $x_i (i = 1, \ldots, 4)$. Then, these messages are passed to function nodes f_1 and f_2 from variable nodes $x_i (i = 1, \ldots, 4)$ as shown in Equation 3.10. When all the messages from the child nodes of f_1, $p_{x_1 \to f_1}(x_1 = 0)$, $p_{x_1 \to f_1}(x_1 = 1)$, $p_{x_2 \to f_1}(x_2 = 0)$, $p_{x_2 \to f_1}(x_2 = 1)$, have arrived, it can compute the result based on the valid configurations of f_1 according to Equation 3.11 and send the result as a message to variable node s, which passes this message to function node f_2. When all the messages from the child nodes of f_2, $p_{x_4 \to f_2}(x_4 = 0)$, $p_{x_4 \to f_2}(x_4 = 1)$, $p_{s \to f_2}(s = 0)$, $p_{s \to f_2}(s = 1)$, have arrived, it

[2]The conditional probability $p(x = 0|y) = 1 - p(x = 1|y)$.

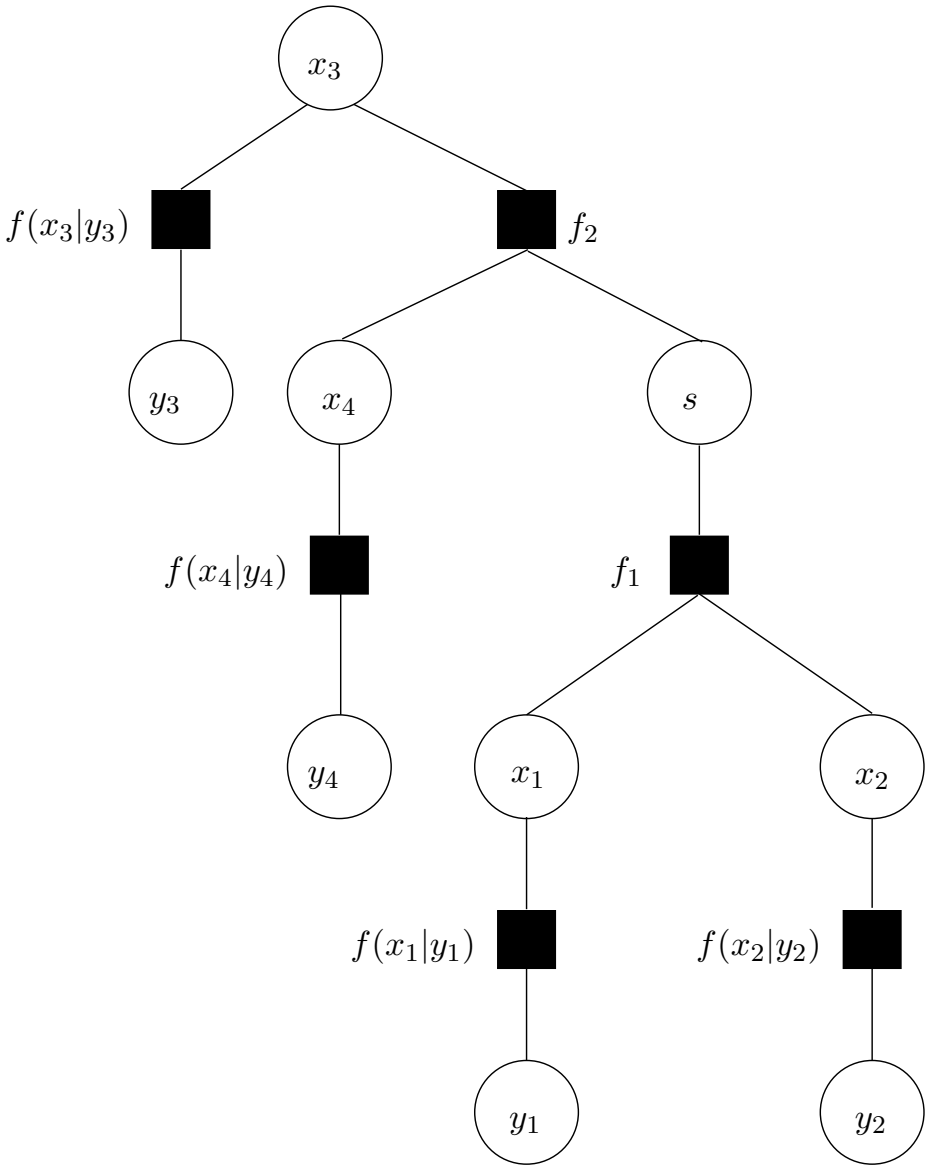

Figure 3.6. A tree representation for the factor graph shown in Figure 3.5.

can compute the result based on the valid configurations of f_2 according to Equation 3.12 and send the result, extrinsic information, as a message to variable node x_3. Finally, when all the messages from the child nodes of x_3 have arrived, the decoded probabilities of x_3 to be '0' and '1' that is the result of the marginal function can be calculated using Equation 3.13 by combining the channel information and extrinsic information for x_3.

$$\begin{aligned}
p_{x_1\to f_1}(x_1=0) &= p(x_1=0|y_1)\\
p_{x_1\to f_1}(x_1=1) &= p(x_1=1|y_1)\\
p_{x_2\to f_1}(x_2=0) &= p(x_2=0|y_2)\\
p_{x_2\to f_1}(x_2=1) &= p(x_2=1|y_2)\\
p_{x_3\to f_2}(x_3=0) &= p(x_3=0|y_3)\\
p_{x_3\to f_2}(x_3=1) &= p(x_3=1|y_3)\\
p_{x_4\to f_2}(x_4=0) &= p(x_4=0|y_4)\\
p_{x_4\to f_2}(x_4=1) &= p(x_4=1|y_4)
\end{aligned} \tag{3.10}$$

$$\begin{aligned}
p_{f_1\to s}(s=0) &= p_{x_1\to f_1}(x_1=0)*p_{x_2\to f_1}(x_2=0)+\\
&\quad\ p_{x_1\to f_1}(x_1=1)*p_{x_2\to f_1}(x_2=1)\\
p_{f_1\to s}(s=1) &= p_{x_1\to f_1}(x_1=0)*p_{x_2\to f_1}(x_2=1)+\\
&\quad\ p_{x_1\to f_1}(x_1=1)*p_{x_2\to f_1}(x_2=0)
\end{aligned} \tag{3.11}$$

$$\begin{aligned}
p_{f_2\to x_3}(x_3=0) &= p_{s\to f_2}(s=0)*p_{x_4\to f_2}(x_4=0)+\\
&\quad\ p_{s\to f_2}(s=1)*p_{x_4\to f_2}(x_4=1)\\
&= p_{f_1\to s}(s=0)*p_{x_4\to f_2}(x_4=0)+\\
&\quad\ p_{f_1\to s}(s=1)*p_{x_4\to f_2}(x_4=1)\\
p_{f_2\to x_3}(x_3=1) &= p_{s\to f_2}(s=0)*p_{x_4\to f_2}(x_4=1)+\\
&\quad\ p_{s\to f_2}(s=1)*p_{x_4\to f_2}(x_4=0)\\
&= p_{f_1\to s}(s=0)*p_{x_4\to f_2}(x_4=1)+
\end{aligned}$$

$$p_{f_1 \to s}(s = 1) * p_{x_4 \to f_2}(x_4 = 0) \tag{3.12}$$

$$\begin{aligned} p(x_3 = 0) &= p_{f_2 \to x_3}(x_3 = 0) * p(x_3 = 0|y_3) \\ p(x_3 = 1) &= p_{f_2 \to x_3}(x_3 = 1) * p(x_3 = 1|y_3) \end{aligned} \tag{3.13}$$

If we need to know $p(x_4 = 0)$ and $p(x_4 = 1)$, a similar process is used and even the intermediate result $p_{f_1 \to s}(s = 0)$ and $p_{f_1 \to s}(s = 1)$ can be reused. If we need to calculate $p(x_1 = 0)$ and $p(x_1 = 1)$, a similar process is again used. However, this time, the messages passed from and to node s are different. In general, there are messages passed in both directions of an edge. In the process, there are two kinds of operations, one is multiply to get the product terms and the other is sum and this is why it is called the sum-product algorithm.

3.2.2 The Sum Product Update Rules

The previous subsection discusses how to compute a marginal function. However, in many cases, we are interested in computing multiple marginal functions. Of course, we can compute them one by one using the method mentioned in the previous subsection. Since many intermediate results in computing one marginal function can be reused by other marginal functions as shown in the example used in the previous subsection, this method is not a good one because we can compute all of the marginal functions at the same time on the factor graph. In this case, there are no parent and child nodes. Each node v waits on the messages from all but one of its neighbor nodes. Then, it computes a message to be sent to the remaining neighbor node (temporarily regarded as a parent node). Just as described before, if v is a variable node, it simply computes the product of all the received messages. If v is a function node, it needs to form the product of the received message according to the specified function. As a result, the sum-product update rules can be expressed by Figure 3.7, Equation 3.14, and Equation 3.15. When variable node x has received all the messages from the function nodes $h_1, \ldots, h_m$, it simply computes the product of these messages and then passes the result to its remaining neighbor function node f just as shown in Equation 3.14. When function node f has received all the messages from the variable nodes $y_1, \ldots, y_n$, besides computing the product of these messages, it also need to marginalize on variable x according to the function f before the result is passed to its remaining neighbor function node x just as shown in Equation 3.14.

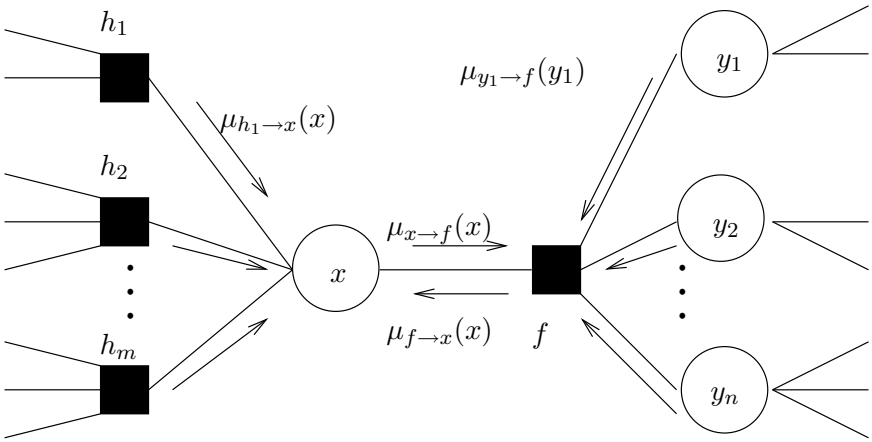

Figure 3.7. Sum product algorithm.

For probability propagation networks, all the messages are probabilities and the functions are indicator functions. As a result, the variable node x just generates the products of the incoming probabilities while the function node f generates the products of the incoming probabilities and then sums up the valid products defined by the indicator function according to variable x. Noticing from Equation 3.14, we know that for a variable node with degree 2, the message is just passed on. The messages for a leaf variable node x are defined by Equation 3.16 and the messages for a function node f are defined by Equation 3.17. Also, the marginal function for a variable node x is calculated in the same way as the previous subsection by the product of all the incoming messages to the variable node. From Equation 3.14, we know the product of all the incoming messages equals the product of the two messages of opposite directions on any edge adjacent to the variable node. Thus the marginal function for variable node x, which is adjacent to a function node f, can be calculated by Equation 3.18.

$$\mu_{x \to f}(x) = \prod_{i=1}^{m} \mu_{h_i \to x}(x) \tag{3.14}$$

$$\mu_{f \to x}(x) = \sum_{\sim\{x\}} \left(f(x, y_1, \cdots, y_n) \prod_{j=1}^{n} \mu_{y_j \to f}(y_j) \right) \tag{3.15}$$

$$\mu_{x \to f}(x) = 1 \tag{3.16}$$

$$\mu_{f\to x}(x) = f(x) \tag{3.17}$$

$$\begin{aligned} g(x) &= \mu_{f\to x}(x)\prod_{i=1}^{m}\mu_{h_i\to x}(x) \text{ (Using Equation 3.14)} \\ &= \mu_{f\to x}(x)\mu_{x\to f}(x) \end{aligned} \tag{3.18}$$

3.2.3 Message Passing Schedule

The previous subsection discusses how to calculate the messages. However, we have not discussed how to initiate the updates and how to pass the messages. According to the update rules shown in Equation 3.14 and Equation 3.15, a message depends on the messages that have been sent before and we need to initialize the messages on the graph. A good solution is to assume unit messages on every edge just as the messages being sent out by leaf nodes as shown in Equation 3.16 and Equation 3.17. Another assumption is the synchronized message passing schedule in which we assume that message passing is synchronized with a global clock. Since we need to do the sum product operation step by step in the high-level model, this is a reasonable assumption and it gives us a good model for the structure. However, finding the optimal message passing schedule that uses minimum message transmissions is still not a trivial problem. For circuit speed consideration and the analog implementation that uses a parallel structure, *flooding schedule* is the best. In this schedule, once a new message is generated, it is then passed to its destination and once the result on an edge changes, a new message is generated on that edge. If multiple messages are generated at the same time on the graph, then they are all passed out at the same time. Also, the flooding schedule is a simple message passing schedule for simulation software since there is no order to the update.

For a factor graph with no cycles, it can be viewed as a tree. As a result, it has leaf nodes with degree 1. Since leaf nodes absorb messages, it is certain that when using the flooding schedule, the messages are all absorbed in a finite time and the graph reaches a final stable state. At this state, we can calculate the marginal functions. However, for a graph with cycles, since a node v in a cycle can send out a message and messages can form a message sequence along the cycle and generate a message sending to v to repeat this process again, there might always be messages passing on the graph. As a result, we need to terminate the calculations at some step. For a good factor graph, after some steps, the new message generated on an edge has a value quite near to the value of the old message on the edge and we say that the sum product algorithm converges. When

the sum product algorithm converges and whether it can converge to the desired result is still a hot topic being researched [9], [12], [19], [34], [47], [64], [46].

3.2.4 Using the Sum Product Algorithm on a Decoder

Actually, the bit-wise MAP decision uses the sum-product operation. From Equation 2.35, we know that we are summing the probabilities of valid codewords. Suppose that all the valid codewords construct a valid code space C, then Equation 2.35 can be rewritten as Equation 3.19 in which $[x \in C]$ is a indicator function indicating whether x is a valid codeword or not.

$$\tilde{u}_{kMAP}(y) = max_{u_k} \left[\sum_{\sim\{u_k\}} [x \in C] \, P(x_i|y_i)) \right] \tag{3.19}$$

For trellis coding, all the valid trellis paths T construct the valid code space. As a result, the MAP decision can be further rewritten as Equation 3.20. For trellis coding, trellis paths are constructed by the trellis sections. Each trellis section has four variables, state s_i at time i, state s_{i+1} at time $i+1$, the information bit u_i and the encoded output x_i. As a result, usually the factor graph for a trellis code is shown as Figure 3.8. The factor graph for a conventional trellis shown in Figure 3.8 can be dissected as n identical sections as shown in Figure 3.9. The messages are also shown in Figure 3.9. Also, notice that the state variable node s_i is not visible, it is an auxiliary variable used to construct the factor graph. As a result, it is shown in a double circle to differentiate it from visible variable nodes. Node variable u_i is a leaf variable node. It provides a unity distribution message. As a result, the outgoing message from u_i can be ignored when doing calculations and the marginal function result of u_i is simply the incoming message to u_i. Also, we do not concern the down going messages to variable nodes x_i, y_i and variable node x_i only pass messages. Thus, the channel function nodes and variable nodes y_i can be omitted for simplicity in analysis. According to the sum product algorithm, Equation 3.21, Equation 3.22, and Equation 3.23 can be derived. Compared with Equation 2.40, Equation 2.41, and Equation 2.39, we know that the forward-backward algorithm is an application of using the sum product algorithm on probability propagation networks. For a conventional trellis, s_0 and s_n only has one state that we can use 0 to represent it. As a result, the outgoing messages of s_0 and s_n are shown in Equation 3.24.

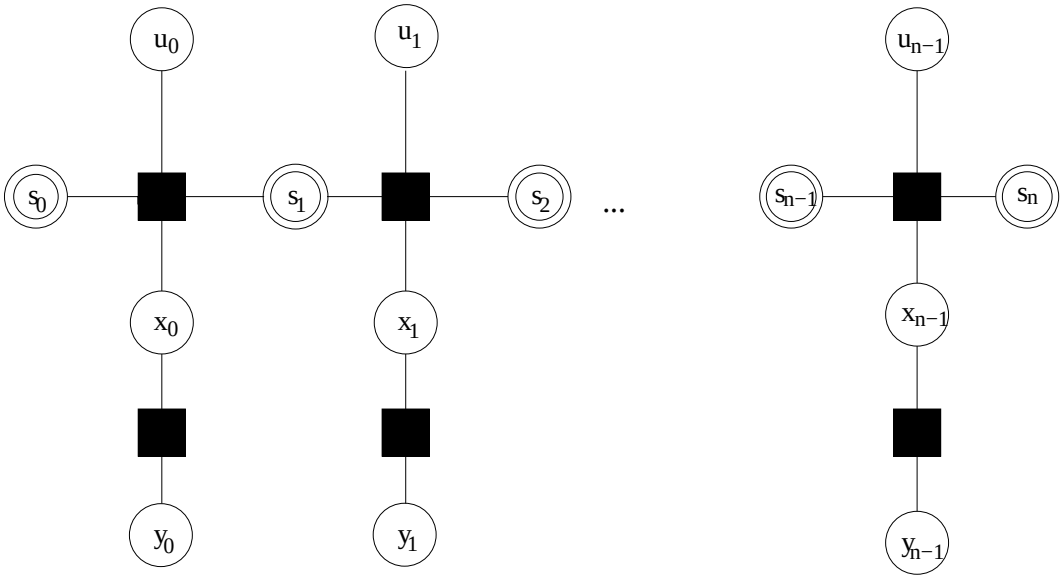

Figure 3.8. Factor graph representation of a conventional trellis.

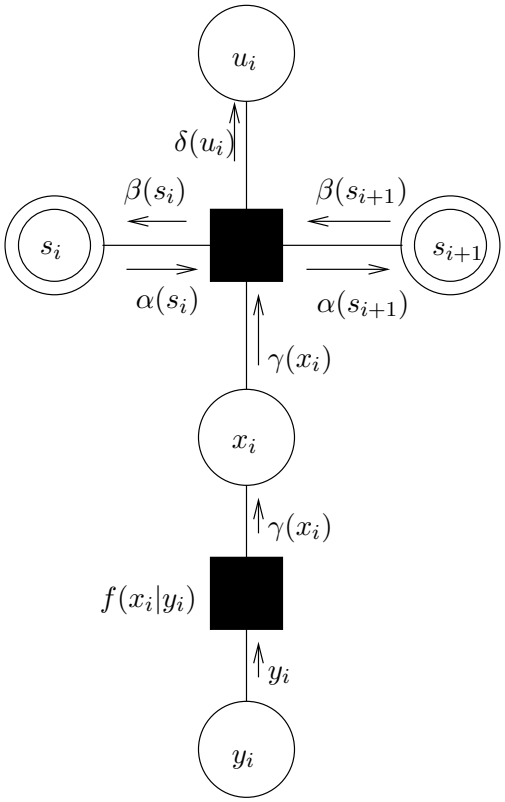

Figure 3.9. One section of the factor graph representation for trellis coding.

$$\tilde{u}_{kMAP}(y) = max_{u_k}\left[\sum_{\sim\{u_k\}}[x \in T]\, P(x_i|y_i))\right] \tag{3.20}$$

$$\alpha(s_{i+1}) = \sum_{s_i}\sum_{u_i}\sum_{x_i}[(s_i, u_i, x_i, s_{i+1}) \in T_i]\alpha(s_i)\gamma(x_i) \tag{3.21}$$

$$\beta(s_i) = \sum_{s_{i+1}}\sum_{u_i}\sum_{x_i}[(s_i, u_i, x_i, s_{i+1}) \in T_i]\beta(s_{i+1})\gamma(x_i) \tag{3.22}$$

$$\delta(u_i) = \sum_{s_i}\sum_{s_{i+1}}\sum_{x_i}[(s_i, u_i, x_i, s_{i+1}) \in T_i]\alpha(s_i)\beta(s_{i+1})\gamma(x_i) \tag{3.23}$$

$$s_0(0) = 1;\ s_n(0) = 1 \tag{3.24}$$

For a tail-biting trellis, the corresponding factor graph is shown in Figure 3.10. According to the sum product algorithm, the initial messages on each edge have uniform probability distributions. Beginning with this initial condition, it is proven that the sum product algorithm for the tail-biting trellis converges to the optimum result [1][3]. Using the compact tail-biting trellis of the extend Hamming (8,4) code shown in Figure 2.11 as an example, the factor graph of the Hamming (8,4) decoder can be derived as shown in Figure 3.11 (The channel function nodes and variable nodes $y_i, (i = 0, \ldots, 7)$ are omitted for simplicity.). Because for variable nodes $x_i, (i = 0, \ldots, 7)$, only the messages going out are what we concerned. As a result, we can combine the messages going out of two adjacent variable nodes x_{2i}, x_{2i+1} and use only one variable node γ_i to substitute the two variable nodes x_{2i}, x_{2i+1} and use a simplified factor graph shown in Figure 3.12. For example, the message coming out of γ_0 is $\gamma_0(00) = f(x_0 = 0|y_0)f(x_1 = 0|y_1)$, $\gamma_0(01) = f(x_0 = 0|y_0)f(x_1 = 1|y_1)$, $\gamma_0(10) = f(x_0 = 1|y_0)f(x_1 = 0|y_1)$, $\gamma_0(11) = f(x_0 = 1|y_0)f(x_1 = 1|y_1)$ in which $f(x_i = 1|y_i)$ and $f(x_i = 0|y_i)$ are calculated according to Equation 2.25. Then, begining with the initial uniform distribution of the α, β values and the γ values provided above, we can calculate the α, β values along the cycle for the next step using Equation 3.21 and Equation 3.22 until the α, β values have converged.

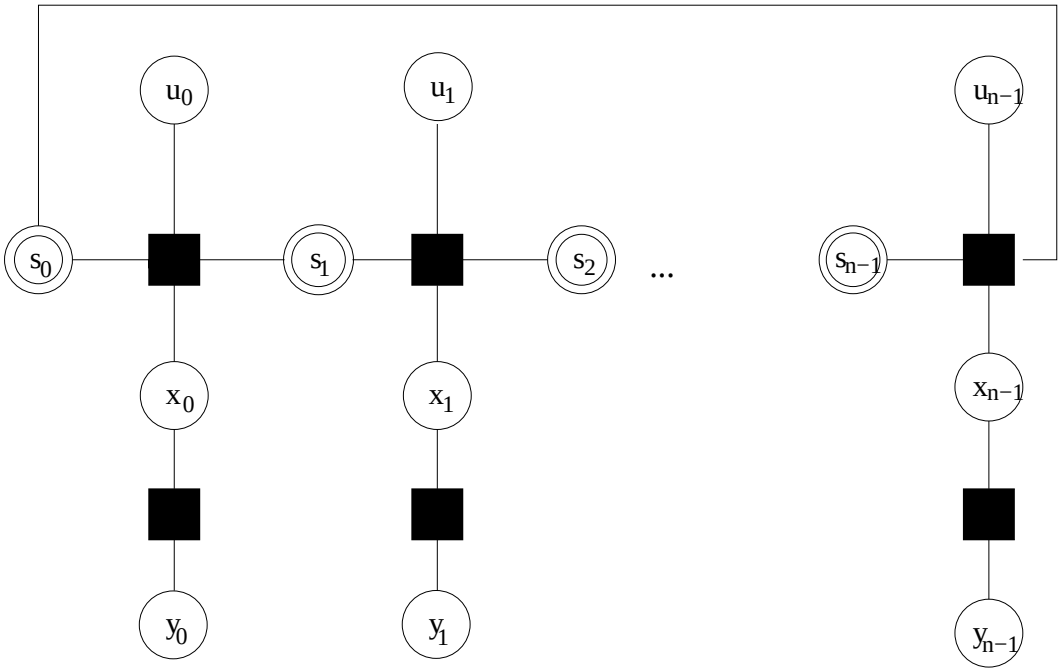

Figure 3.10. Factor graph representation of a tail-biting trellis.

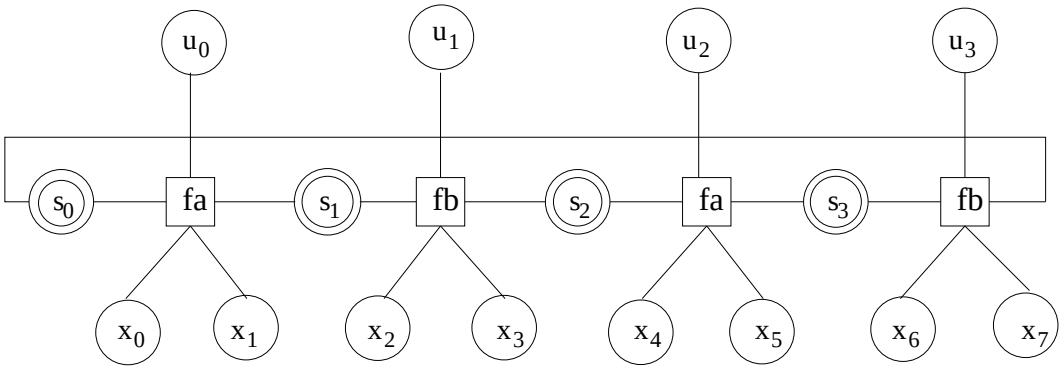

Figure 3.11. Factor graph representation of the Hamming (8,4) decoder.

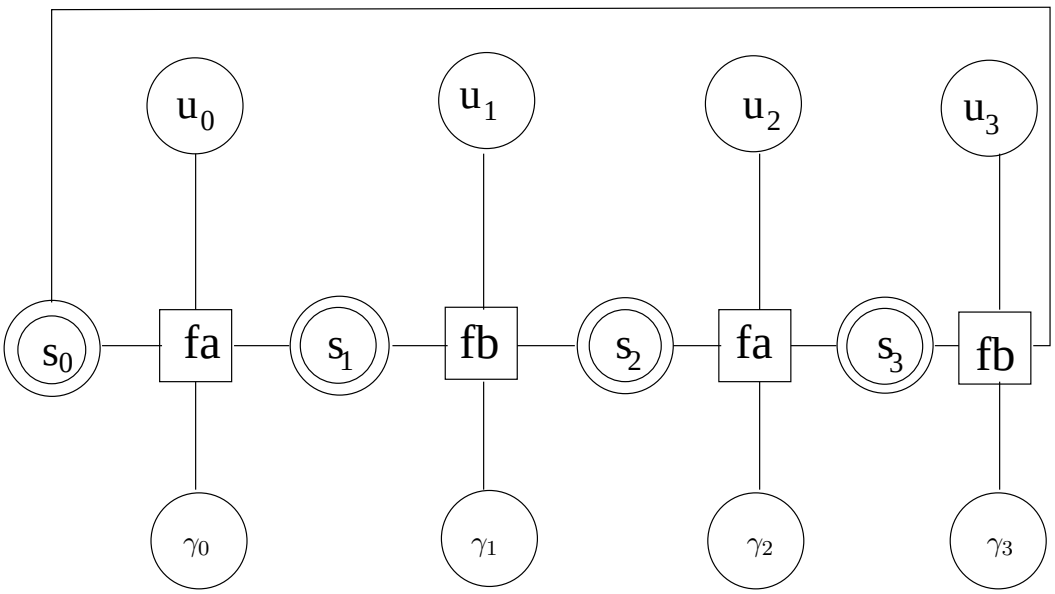

Figure 3.12. Another factor graph representation of the Hamming (8,4) decoder.

Then, we can make the decision on the information bit u_i using Equation 3.23. For example, for the trellis function node between variable node s_0 and s_1 that corresponds to the first trellis section shown in Figure 2.11, the calculation corresponding to Equation 3.21, 3.22 and 3.23 are shown in Equation 3.25, 3.26 and 3.27, which are based on the valid configuration defined by the corresponding trellis section.

$$\begin{aligned}
\alpha(s_1 = 0) &= \alpha(s_0 = 0) * \gamma_0(00) \\
\alpha(s_1 = 1) &= \alpha(s_0 = 0) * \gamma_0(11) \\
\alpha(s_1 = 2) &= \alpha(s_0 = 1) * \gamma_0(01) \\
\alpha(s_1 = 3) &= \alpha(s_0 = 1) * \gamma_0(10)
\end{aligned} \tag{3.25}$$

$$\begin{aligned}
\beta(s_0 = 0) &= \beta(s_1 = 0) * \gamma_0(00) + \beta(s_1 = 1) * \gamma_0(11) \\
\beta(s_0 = 1) &= \beta(s_1 = 2) * \gamma_0(01) + \beta(s_1 = 3) * \gamma_0(10)
\end{aligned} \tag{3.26}$$

$$\delta(u_0 = 0) = \alpha(s_0 = 0) * \gamma_0(00) * \beta_(s_1 = 0) + \alpha(s_0 = 1) * \gamma_0(01) * \beta_(s_1 = 2)$$

$$\delta(u_0 = 1) = \alpha(s_0 = 0) * \gamma_0(11) * \beta_(s_1 = 1) + \alpha(s_0 = 1) * \gamma_0(10) * \beta_(s_1 = 3) \quad (3.27)$$

For iterative decoding, we use the example shown in Figure 3.4. Actually, each parity-check function and its corresponding variables construct a component decoder. In the previous chapter, we have discussed that the essence of iterative decoding is decoding step by step. In each step, the component uses the channel information for the corresponding variables and extrinsic probabilities provided by other component decoders and finally the result is decided by using the channel information for the current bit and all the extrinsic probabilities provided by the component decoders. Let us look at Figure 3.4. The messages provided from a function node to a variable node are generated by the function using the information provided by all the adjacent variable nodes except this variable node. As a result, this message is the extrinsic probability. Thus, the message from a variable node to a function node are the product of the channel information and all the extrinsic information provided by other component decoders, which is just what we need. Also, the marginal function is constructed by the product of the channel information for the current bit and all the extrinsic probabilities provided by the component decoders. This clearly shows that the factor graph model is the exact model for iterative decoding. The structure of an LDPC decoder is similar to the structure shown in Figure 3.4 except an LDPC decoder is deliberately designed so that the sum product algorithm converges to a near optimum result. Interested readers can also verify that other iterative decoders such as Turbo codes and product codes can also be modeled by using sum product algorithms on a factor graph [9], [57], [46].

3.3 Factor Graph Simulation

The previous section discusses that all decoders can be modeled by using the sum product algorithm on factor graphs. This section discusses how to do high-level VHDL simulation of a decoder. Also, since every decoder can be represented by its factor graph representation, we only need its factor graph description for simulation purposes. As a result, an automatic tool is built. Using this tool, only the factor graph representation of the decoder is needed. The tool can generate the VHDL simulation file from the decoder's factor graph representation, which is quite simple.

3.3.1 High Level VHDL Simulation

The simulation of error control decoders has always been a problem due to the large number of simulation cycles required. For example, if the bit error rate is 10^{-5} and 100 errors need to be discovered before the bit error rate is calculated, then 10^7 simulations are needed. Even for a digital circuit, using Verilog or VHDL to simulate the circuit at the register transfer level takes a long time. For an analog circuit, the situation is worse. Spice is an accurate circuit simulation tool. However, one simulation of the extended Hamming (8,4) decoder shown in Figure 2.11 takes 73 second on a Sun Sparc60 workstation. Thus, 10^7 simulations of the extended Hamming(8,4) decoder by using Spice requires about 23 years. As a result, a high level simulation method that can do simulation more efficiently is needed.

For high-level simulation, some researchers use MATLAB to do an exhaustive search over all the valid codewords to find the bit error rate. This method is quite time consuming. The simulation of the extended Hamming (8,4) decoder shown in Figure 2.11 takes 7 days for SNR=7. Also, a main purpose of the high level simulation is to verify whether the structure of the circuit is correct while this method cannot verify this.

From the previous description, we know that a decoder can be modeled by using the sum product algorithm on a factor graph. Also, for the sum product algorithm, we can use a synchronized flooding message passing schedule. As a result, we can use a global clock and a VHDL behavioral module to model each function node and variable node. The initial messages are all set to a unity probability distribution according to the rule. For every clock cycle, if any message coming into the module has changed, then all the output messages of the module are calculated and sent out. Using this method, the sum product algorithm is modeled exactly and the structure of the circuit can be verified. For decoders, the ratio between the probabilities is what we need. As a result, the output messages for each module can be normalized to avoid small probability calculations.

In the simulation, all the probabilities passing between modules are real numbers. One might think to use Verilog-A or VHDL-AMS to do the simulation. However, they are too complicated to be simulated efficiently. Standard VHDL allows the use of real numbers and it is enough to do behavioral level simulation. As a result, standard VHDL is chosen to do simulation accurately and efficiently. For the simulation of the extended Hamming (8,4) decoder shown in Figure 2.11, it takes no more than 1 hour for SNR=7 using this method.

3.3.2 Normal Graph

From the previous description, we know that leaf nodes absorb messages, variable nodes with degree 2 pass messages from one function node to another function node, variable nodes with degree n ($n > 2$) generate the product of $n - 1$ incoming messages and send it out on the remaining edge. Thus, a variable node with degree n ($n > 2$) performs as an equal gate. The function of an equal gate for a variable with two states 0 and 1 is shown in Equation 2.44 and the function of equal gates for variables with more than two states can be derived similarly (In Equation 2.44, the result is normalized.). As a result, variable nodes with degree n ($n > 2$) can be split into n variable nodes with degree 2 and an equal gate function node that is connected to these n variable nodes as shown in Figure 3.13. For variable nodes with degree 2, it can be omitted since it only passes messages. This allows direct connections between function nodes, changing a factor graph to a normal graph defined by Forney [30][31]. After doing these two steps, there are only variable nodes with degree 1. For variable nodes with degree 1, they only provide a constant message and absorb messages. Thus, there is no calculation needed for a variable node with degree 1. As a result, all calculations are implemented by function nodes. Therefore, for the high-level VHDL simulation, we need to provide modules only for the function nodes.

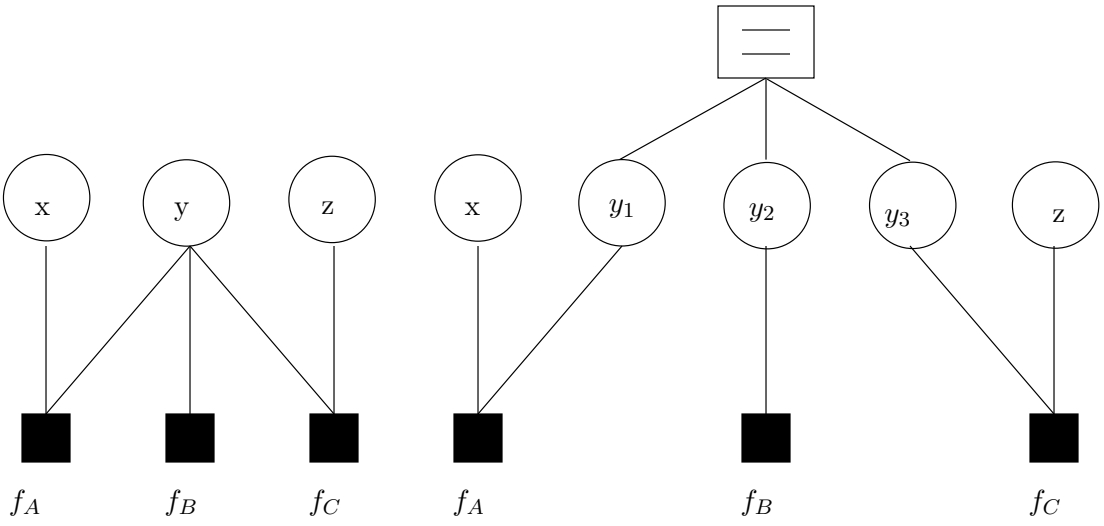

Figure 3.13. Variable node splitting for a variable node with degree more than 2.

3.3.3 Factor Graph Description

The previous section states that a decoder can be represented by its factor graph description. Also, the decoder can also be represented by a normal graph using the method mentioned in the previous subsection. For a normal graph, if all the function nodes are implemented by circuits, then by connecting the circuits representing the function nodes according to the normal graph connections, the decoder can be realized. As a result, the function node is the center for both implementation and simulation.

For a function node, it can always be described by an indication function. For example, an equal gate function node with connection to three variable nodes x, y, z can be described by Equation 3.28 while an XOR function node with connection to three variable nodes x, y, z can be described by Equation 3.29 in which every string separated with a semicolon shows a valid configuration. The factor graph view is the same and the structure of the graph is shown in Figure 3.14. Notice that the factor graph is defined not only by the structure, but also by the local functions. Thus, even though their structures are the same, the indicator function shown in Equation 3.28 and Equation 3.29 differ for the equal gate and the soft XOR gate.

$$[x, y, z] = [0, 0, 0; 1, 1, 1] \tag{3.28}$$

$$[x, y, z] = [0, 0, 0; 0, 1, 1; 1, 0, 1; 1, 1, 0] \tag{3.29}$$

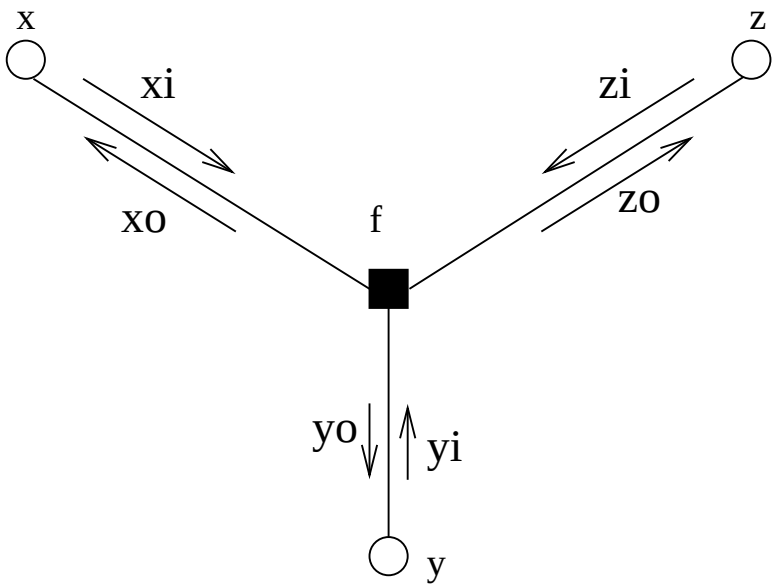

Figure 3.14. Factor graph structure of the equal gate and XOR gate.

Notice that the messages passed in each edge are bidirectional, in order to describe the operation of the function node, messages in each direction need to be defined as shown in Figure 3.14. Also, notice that in each message, there are actually two probabilities, one to represent the probability of being 0 and the other to represent the probability of being 1. As a result, the high-level VHDL description of the operations of the equal gate function node and the XOR function node in every clock cycle according to the factor graph and the sum-product algorithm definition are shown in the following in which xi(0) represents the probability of being 0 in message xi while xo(0) represents the probability of being 0 in message xo, etc. temp_xo, temp_x, etc. are temporary variables used to simplify the descriptions and operations. In the description, normalization is used to ensure that the sum of the output probabilities equals 1 and the normalization does not change the ratio between the probabilities. A complete high-level VHDL description of the equal gate function node and XOR function node can be found in Appendix C.

Equal gate:

```
temp_xo(0):=yi(0)*zi(0);
temp_xo(1):=yi(1)*zi(1);
temp_x:=temp_xo(0)+temp_xo(1);
xo(0)<=temp_xo(0)/temp_x;
xo(1)<=temp_xo(1)/temp_x;
temp_yo(0):=xi(0)*zi(0);
temp_yo(1):=xi(1)*zi(1);
temp_yo:=temp_yo(0)+temp_yo(1);
yo(0)<=temp_yo(0)/temp_y;
yo(1)<=temp_yo(1)/temp_y;
temp_zo(0):=xi(0)*yi(0);
temp_zo(1):=xi(1)*yi(1);
temp_z:=temp_zo(0)+temp_zo(1);
zo(0)<=temp_zo(0)/temp_z;
zo(1)<=temp_zo(1)/temp_z;
```

XOR function node:

```
temp_xo(0):=yi(0)*zi(0)+yi(1)*zi(1);
temp_xo(1):=yi(0)*zi(1)+yi(1)*zi(0);
temp_x:=temp_xo(0)+temp_xo(1);
xo(0)<=temp_xo(0)/temp_x;
xo(1)<=temp_xo(1)/temp_x;
temp_yo(0):=xi(0)*zi(0)+xi(1)*zi(1);
temp_yo(1):=xi(0)*zi(1)+xi(1)*zi(0);
temp_yo:=temp_yo(0)+temp_yo(1);
yo(0)<=temp_yo(0)/temp_y;
yo(1)<=temp_yo(1)/temp_y;
temp_zo(0):=xi(0)*yi(0)+xi(1)*yi(1);
temp_zo(1):=xi(0)*yi(1)+xi(1)*yi(0);
temp_z:=temp_zo(0)+temp_zo(1);
zo(0)<=temp_zo(0)/temp_z;
zo(1)<=temp_zo(1)/temp_z;
```

Actually, the equal gate function node and the XOR function node discussed above can be regarded as simple codes with length 3. The code defined by the equal gate function node has only two valid codewords 000 and 111. The code defined by the XOR function node has four valid codewords 000,011,101,110 that are defined by the function $x + y + z = 0$ in which '+' represents the XOR operation in the binary field. The input probabilities to the function node, $xi(0)$, $xi(1)$, $yi(0)$, $yi(1)$, $zi(0)$, and $zi(1)$ are provided by the received value and the channel characteristics using Equation 2.25. The outputs of the soft decision decoder are given by $xo(0)$, $xo(1)$, $yo(0)$, $yo(1)$, $zo(0)$, and $zo(1)$ using the operations provided above. $xo(0)$ means the probability of bit x to be '0' given by the decoder and $xo(1)$ means the probability of bit x to be '1' given by the decoder. Compared with the operations provided above, the factor graph descriptions of the equal gate function node and the XOR function node shown in Equation 3.28 and Equation 3.29 are much simpler.

For variable nodes with degree 1, its value (the marginal function for this variable) is described by the incoming message for this variable and we do not need to describe it. For the connections, we need to define signals to connect the function nodes. In order to connect a signal to a function module, we need to define the ports that are the interface of a function module. Of course, all the ports and signals have types to show which kind of message it can carry. For decoders, all the probabilities passed between the function modules are real numbers between 0 and 1. Thus, for the types of messages, we only need to define how many probabilities are included in the message. Also, in the factor graph, messages can be passed in two opposite directions on the edge. As a result, the ports and signals should be bi-directional. However, from Figure 3.8, we know that for some nodes such as x_i, we do not need to know the value of the incoming message toward x_i and for the leaf variable nodes such as u_i, they provide a uniform distribution probability so that the message coming out of u_i can be omitted. As a result, for the description of a decoder, we provide an optional direction "in" and "out" for port and signal descriptions to simplify the simulation process and minimize the derived circuit. Also, for the description of a complex factor graph, it is better to describe the connection hierarchically. As a result, we can define some function module by a structural description of how it is implemented by connecting some low level function modules. Just as many programming languages, if the low level function modules are described in different files, these files should be described as "include FileName" in the description of the high-level

function modules. For a detailed description of the language that we designed to describe a factor graph, please see Appendix A.

3.3.4 Automatic Simulation from a Factor Graph Description

The previous subsections describe how to use a VHDL file to do high level simulation. Also, the decoder can be described by its normal graph description and there is a quite direct relationship between the normal graph description and the VHDL behavioral level description. As a result, we can develop an automatic tool to translate the factor graph description into the VHDL behavioral level description.

However, in order to do simulation, only having the description of the decoder is not enough. We also need to provide the simulation environment. The encoder is needed to be described and we need to connect the encoder and decoder as shown in Figure 3.15. In many conditions, the encoder is not described by a factor graph and we need to provide some operations and special functions needed for the description of the encoder. Also, in order to describe some large decoders that have duplicate structures and conditional structures. Loop control and conditional control statements should be provided.

All in all, our goal is that the factor graph description of the decoder and the description of the simulation environment should be as simple as possible to make the design process easy. As a result, the automatic simulation of the decoder is not easy and many problems must be solved. The language used for the description of both the factor graph and the encoder are shown in Appendix A and the extended Hamming (8,4,4) decoder is used as an example.

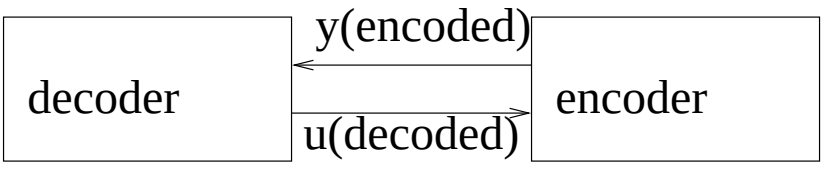

Figure 3.15. Using an encoder as the simulation environment.

CHAPTER 4

AUTOMATIC SYNTHESIS

The previous chapter describes factor graphs and the sum-product algorithm. We also show that error control coding can be interpreted as operations of the sum-product algorithm on probability propagation networks that is a kind of factor graph [17], [35], [2]. As a result, we can use high-level VHDL simulation based on a decoder's factor graph description to verify the structure of the decoder. Moreover, a tool has been built that can generate the VHDL simulation file automatically from a decoder's factor graph representation that is quite simple. This chapter describes how an analog decoder is realized by using basic building blocks, how these basic building blocks are realized by circuit implementation, and how to connect these building blocks. We also show that all the basic building blocks can be constructed by a few basic cells. As a result, a cell library composed of these cells is built. Moreover, an automatic synthesis tool is produced. Using this tool, one can build the circuit for an analog decoder by only providing the factor graph description of the decoder with little or no analog circuit design knowledge.

4.1 Basic Building Block

The previous chapter shows that error control coding systems can be implemented by using the sum-product algorithm on probability propagation networks in which all the messages passing from one node to another have the meaning of probabilities or probability density functions. As a result, the basic building block has the form shown in Figure 4.1. In Figure 4.1, the building block computes a discrete probability mass function p_z from the discrete probability mass functions p_x and p_y as follows. Let $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$ be finite sets. Let p_x and p_y be the input probability mass functions defined on the sets (alphabets) $\mathcal{X}$ and $\mathcal{Y}$, respectively. Let p_z be the output probability mass function on $\mathcal{Z}$ defined by Equation 4.1.

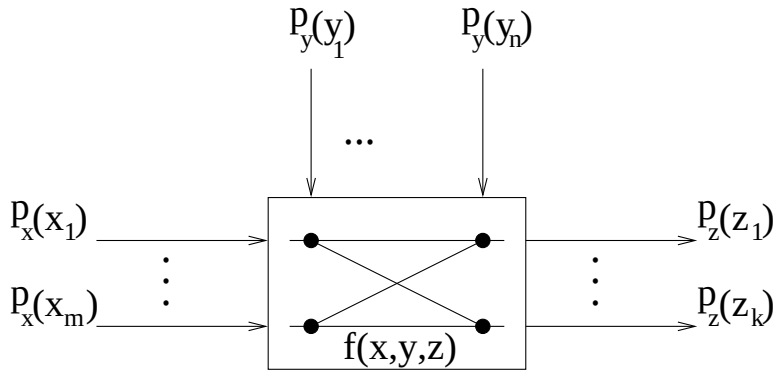

Figure 4.1. The building block of the probability propagation network.

$$p_z(z) = \gamma \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p_x(x) p_y(y) f(x, y, z), \forall z \in \mathcal{Z}, \tag{4.1}$$

where f is a function from $\mathcal{X} \times \mathcal{Y} \times \mathcal{Z}$ into {0,1} and where γ is an appropriate scale factor that does not depend on z. The scale factor γ is required to yield a probability distribution $p_z(z)$ at the output whose sum is $\sum_{i=1}^{k} p_z(z_i) = 1$. As a result, this building block implements a marginalization function.

For example, for a function node that has connections with three variable nodes as shown in Figure 3.14, three building blocks are used to implement it. One building block has xi, and yi as inputs and zo as output. One building block has yi, and zi as inputs and xo as output. The other building block has xi, and zi as inputs and yo as output.

For a hidden function f that has more than three variables, it is partitioned into a series of sum-products of two variables so that the building block of Figure 4.1 can still be used. For example, for hidden function $g(xa, xb, xc, xd)$, we can use the following equations to calculate the marginal functions.

$$g_1(xd) = \sum_{xa,xb,xc} g(xa, xb, xc, xd) = \sum_{xc} \sum_{xa,xb} g(xa, xb, xc, xd) \tag{4.2}$$

$$g_1(xc) = \sum_{xa,xb,xd} g(xa, xb, xc, xd) = \sum_{xd} \sum_{xa,xb} g(xa, xb, xc, xd) \tag{4.3}$$

As a result, the implementation is shown in Figure 4.2. The form of the basic building block is still the same. Also, the intermediate result can be used by several other blocks as input to decrease the circuit complexity.

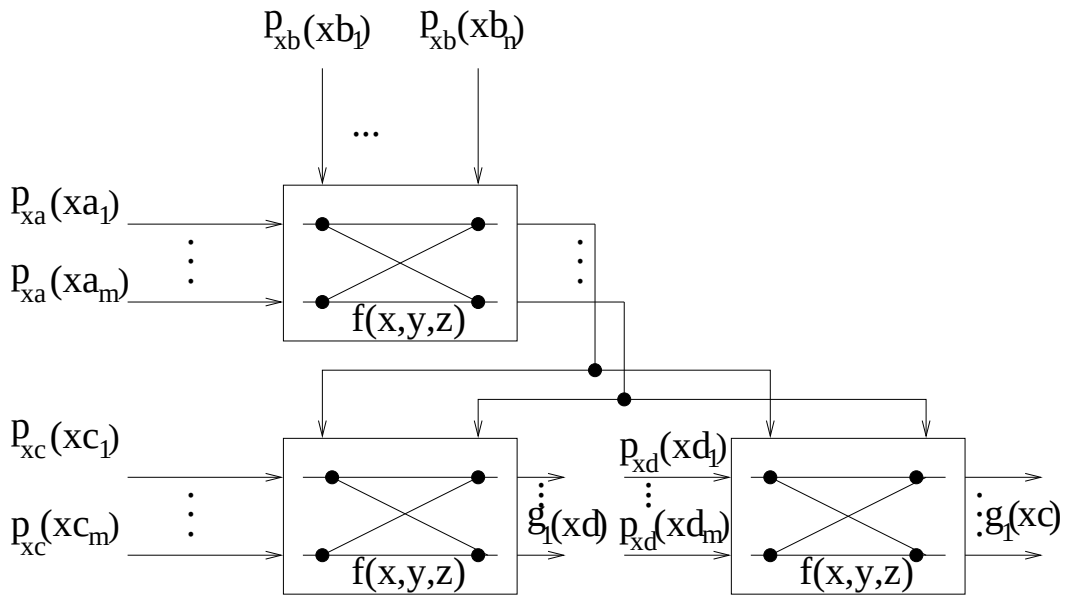

Figure 4.2. Using several building blocks to implement the hidden function with more than three variables.

4.2 Circuit Implementation of the Basic Building Block

The basic building block performs two operations. It needs to generate the product terms and then sum the product terms according to the hidden function f. Let us consider the product operation first.

It has been revealed that for MOS transistors working under the weak inversion region, the following equations exist where I_{0n} and I_{0p} are process-dependent constants for NMOS transistors and PMOS transistors respectively.

$$\begin{aligned} I_{0n} &= \frac{2\mu_n C'_{ox} U_T^2}{\kappa} e^{\frac{-\kappa V_{T0n}}{U_T}} \\ I_{DS} &= I_{0n} \frac{W}{L} e^{\frac{\kappa V_G - V_S}{U_T}} \quad for\, V_{DS} > 4U_T\,(saturation) \end{aligned} \tag{4.4}$$

$$\begin{aligned} I_{0p} &= \frac{2\mu_p C'_{ox} U_T^2}{\kappa} e^{\frac{\kappa V_{T0p}}{U_T}} \\ I_{DS} &= I_{0p} \frac{W}{L} e^{\frac{\kappa(V_W - V_G) - (V_W - V_S)}{U_T}} \quad for\, V_{DS} > 4U_T\,(saturation) \end{aligned} \tag{4.5}$$

For the circuit shown in Figure 4.3 that is composed of N identical NMOS transistors, according to the translinear theory [21], [60], [23], we can form a loop as shown in

Figure 4.3 by the dotted line of the gate-source voltages of the N NMOS transistors, $N/2$ of the transistors in clock-wise (CW) direction and the others in counter-clock-wise (CCW) direction. Using Kirchnoff's voltage law, Equation 4.6 is true.

$$\sum_{CW} V_{GS_i} = \sum_{CCW} V_{GS_i} \tag{4.6}$$

Combining Equation 4.4 and Equation 4.6, we obtain Equation 4.7. Actually, Equation 4.7 is the result of using translinear theory on the circuit shown in Figure 4.3. Similar analysis on PMOS transistors generates a similar result.

$$\begin{aligned} e^{\sum_{CW} V_{GS_i}} &= e^{\sum_{CCW} V_{GS_i}} \\ \prod_{CW} e^{V_{GS_i}} &= \prod_{CCW} e^{V_{GS_i}} \\ \prod_{CW} I_{DS_i} &= \prod_{CCW} I_{DS_i} \end{aligned} \tag{4.7}$$

The circuit shown in Figure 4.4 is the fundamental circuit used in probability propagation networks. Let us define $I_x = \sum_{i=1}^{m} I_{x,i}$, $I_y = \sum_{j=1}^{n} I_{y,j}$, and $I_z = \sum_{i=1}^{m} \sum_{j=1}^{n} I_{i,j}$. Let $V_{x,i}$ and $V_{y,j}$ denote the potentials at the input terminal for $I_{x,i}$ and $I_{y,j}$, respectively. Then, we have the following equations.

$$\begin{aligned} I_{i,j}/I_{x,i} &= I_{i,j} / \sum_{l=1}^{n} I_{i,l} \\ &= I_{0n} \frac{W}{L} e^{\frac{\kappa V_{y,j} - V_{x,i}}{U_T}} / \sum_{l=1}^{n} I_{0n} \frac{W}{L} e^{\frac{\kappa V_{y,l} - V_{x,i}}{U_T}} \end{aligned}$$

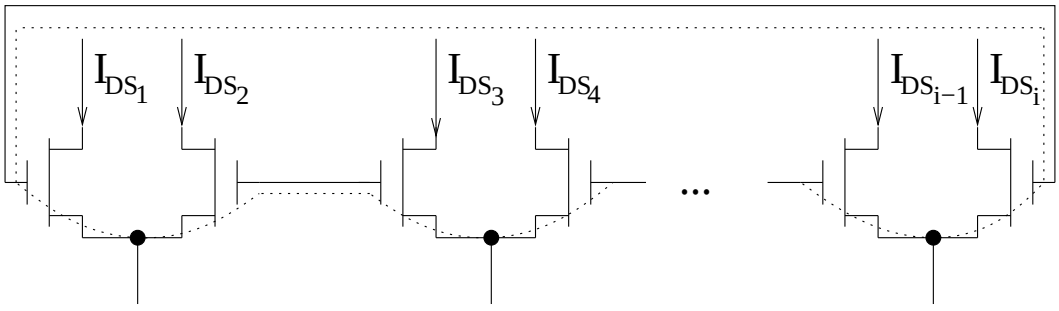

Figure 4.3. A simple translinear loop using NMOS transistors working under subthreshold region.

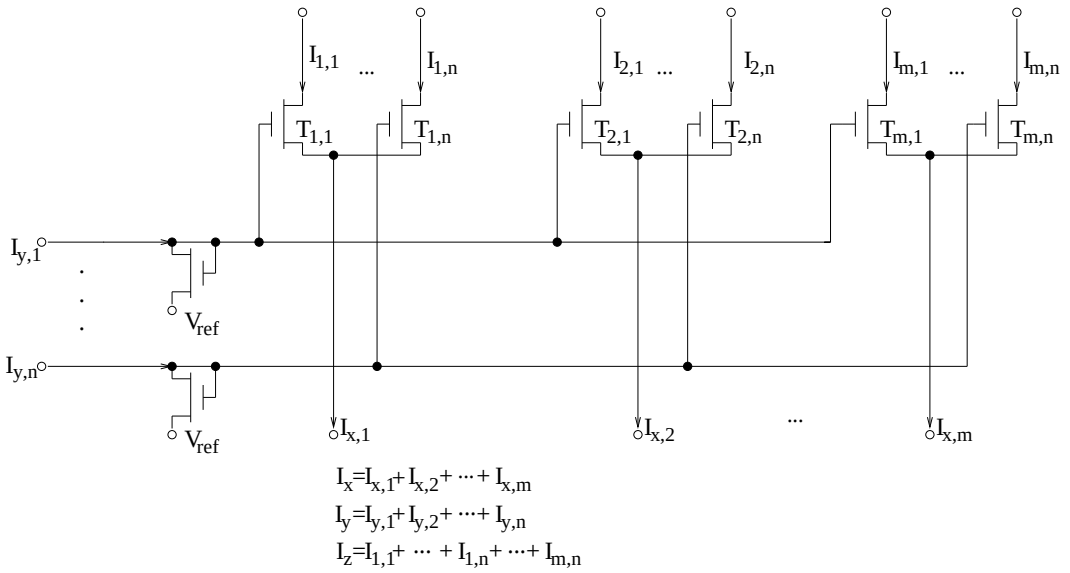

Figure 4.4. Fundamental circuit.

$$= e^{\frac{\kappa V_{y,j}}{U_T}} / \sum_{l=1}^{n} e^{\frac{\kappa V_{y,l}}{U_T}} \tag{4.8}$$

$$\begin{aligned} I_{y,j}/I_y &= I_{y,j} / \sum_{l=1}^{n} I_{y,l} \\ &= I_{0n} \frac{W}{L} e^{\frac{\kappa V_{y,j} - V_{ref}}{U_T}} / \sum_{l=1}^{n} I_{0n} \frac{W}{L} e^{\frac{\kappa V_{y,l} - V_{ref}}{U_T}} \\ &= e^{\frac{\kappa V_{y,j}}{U_T}} / \sum_{l=1}^{n} e^{\frac{\kappa V_{y,l}}{U_T}} \end{aligned} \tag{4.9}$$

Combining Equation 4.8 and Equation 4.9 yields Equation 4.10. Noticing that in probability propagation networks, currents are used to represent probabilities. The ratio of $I_{y,j}$ to I_y represents the probability of information from the Y direction to be equal to the jth signal and $P_y(j)$ is used to represent this probability as shown in Equation 4.11.

$$I_{i,j}/I_{x,i} = I_{y,j}/I_y \tag{4.10}$$

$$I_{i,j} = I_{x,i} \frac{I_{y,j}}{I_y} = I_{x,i} P_y(j) \tag{4.11}$$

If $I_z = I_x$, we have the result shown in Equation 4.12. As a result, the circuit shown in Figure 4.4 can implement the product of probabilities. Actually, by forming all $(n-1)m$

translinear loops in the circuit, writing corresponding equations according to Equation 4.7 and summing all these equations together, we obtain Equation 4.12. Thus, the circuit shown in Figure 4.4 can also be explained using tranlinear theory.

$$I_{i,j}/I_z = (I_{x,i}/I_x)P_y(j) = P_x(i)P_y(j) \tag{4.12}$$

Now, current is used to represent the probabilities. The sum of probabilities can be easily implemented by connecting wires together and using KCL.

For the circuit shown in Figure 4.4, we can observe it in a different view. Instead of observing it in current view, we can observe it in a voltage view. Since $I_{y,j} = I_y P_y(j)$ and I_y is the total current for all the y input and it is a constant current that represent probability 1, using Equation 4.4 and noticing that the source voltage for the transistor V_{ref} shown in Figure 4.4 is a constant, $V_{y,j} = \frac{U_T}{\kappa} \ln P_y(j) + const$ can be derived and the voltage difference is expressed by Equation 4.13. Now, instead of calculating the probabilities directly, we can consider the log-domain probability ratios, which sometimes makes analysis easier. Also, notice that U_T is temperature dependent. As a result, the thermal effect may exist if voltage connection is used and we discuss this in Section 4.4.

$$V_{y,j} - V_{y,i} = \frac{U_T}{\kappa}(\ln P_y(j) - \ln P_y(i)) = \frac{U_T}{\kappa} \ln \frac{P_y(j)}{P_y(i)} \tag{4.13}$$

4.3 Connecting Building Blocks

In building analog decoders, a large number of the building blocks are needed. Of course, these building blocks need to be connected and we discuss this in this section.

4.3.1 Connecting Building Blocks Using Current Mirrors

Notice that in Figure 4.4, the circuit uses current input and current output. A simple approach to the connection problem is to use current mirrors. Also, using current mirrors, we can duplicate the current many times so that the output of one building block can be used by many other building blocks as input. As a result, the connection of building blocks can be accomplished as shown in Figure 4.5.

4.3.2 Stacking and Folding Building Blocks

Besides using current mirrors to connect building blocks, there are two other schemes, stacking and folding building blocks. By directly using the output of one building block

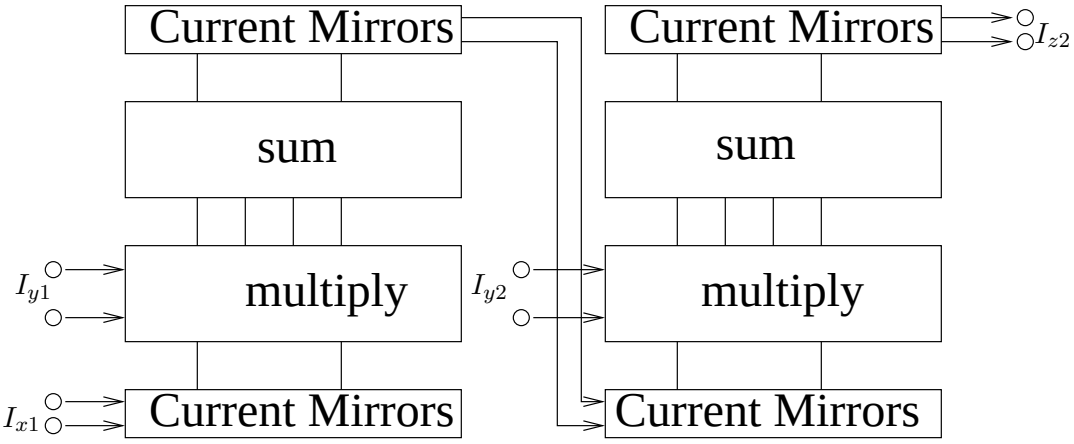

Figure 4.5. Using current mirrors to connect building blocks.

as the input of another building block, we can stack the circuit modules as shown in Figure 4.6. Using this method, the current mirrors between the building blocks can be saved. However, this technique is not possible for low-voltage applications using state-of-the-art silicon technologies.

Another technique is connecting adjacent building blocks by using folded building blocks as shown in Figure 4.7. Using this technique, part of the current mirrors between adjacent building blocks can be saved. However, this technique needs heavy use of PMOS transistors that must be larger than NMOS transistors due to slow mobility of holes, making this technique not a good choice.

4.3.3 Scaling

Another problem that needs to be considered is scaling. Since some of the unused product terms are discarded, the total output current of a building block may be smaller than the total input current if no scaling is used. As the current representing the probability passing between the building blocks may be smaller than the nonideal factors of the circuit, the circuit may not work. As a result, a scaling circuit needs to be used. Gilbert [22] has presented an array-normalize circuit shown in Figure 4.8 that implements the needed function. Actually, the circuit shown in Figure 4.8 is similar to the circuit shown in Figure 4.4 except the circuit shown in Figure 4.8 uses PMOS transistors while the circuit shown in Figure 4.4 uses NMOS transistors. Let us define $I_{in} = \sum_{j=1}^{k} I_{in,j}$. Then, Equation 4.14, which is similar to Equation 4.10, can be derived in which $I_{in,j}$

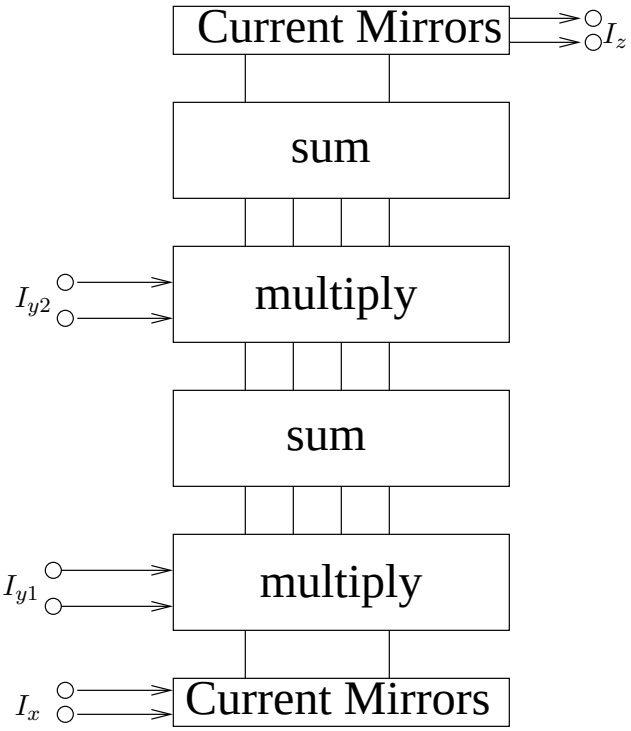

Figure 4.6. Stacking core circuit to connect building blocks.

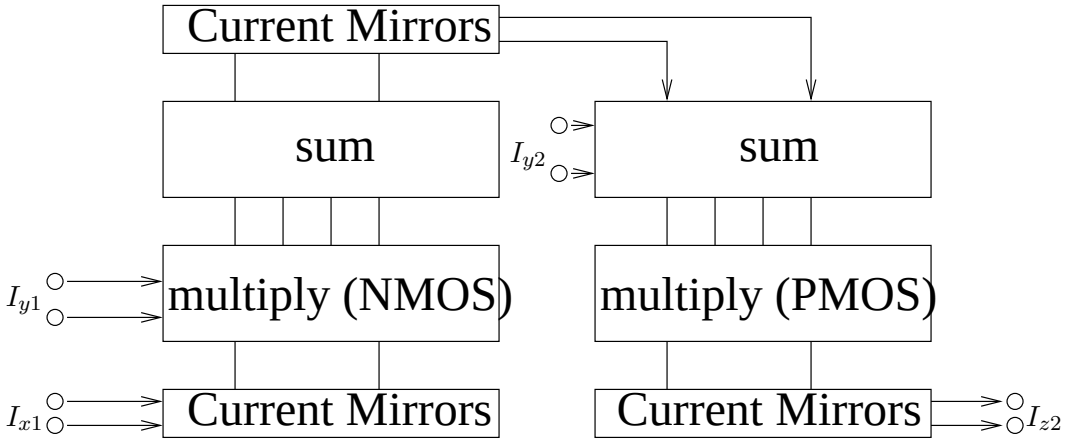

Figure 4.7. Using adjacent n-type and p-type building blocks to construct circuit.

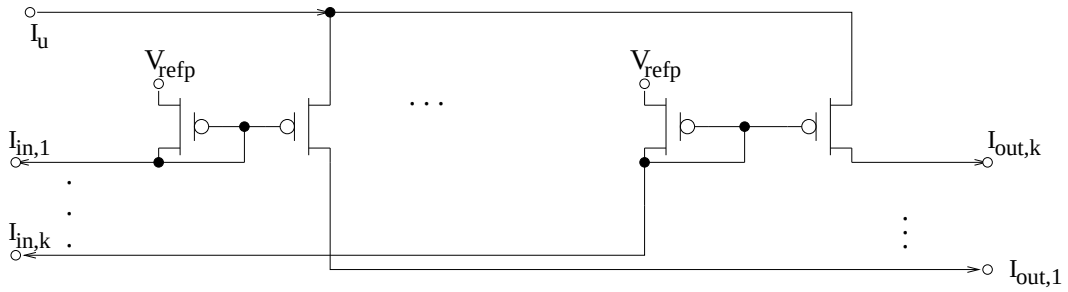

Figure 4.8. A current in, current out normalization circuit.

corresponds to $I_{y,j}$, I_{in} corresponds to I_y, $I_{out,j}$ corresponds to $I_{i,j}$, I_u corresponds to $I_{x,i}$ in Equation 4.10. Notice that the input probability $I_{in,j}/I_{in}$ and the output probability $I_{out,j}/I_u$ are equal and the sum of the output currents is normalized to I_u.

$$I_{in,j}/I_{in} = I_{out,j}/I_u \tag{4.14}$$

One problem that needs to be considered is scaling at the output or input. Scaling at the output has the advantage that the small current is immediately normalized to a nominal level after the operation that causes the current loss and therefore speeds up the circuit. As a result, scaling at the output is chosen.

Of course, scaling at every stage is not necessary because one building block can not make the output current too low. Scaling after every two or three building blocks is possible. However, it is highly application specific and has to be investigated for every code. Also, scaling at every stage speeds up the circuit and makes the circuit more robust against noise and leakage current, which is quite important for a circuit working in the subthreshold region. As a result, scaling at every stage is generally used.

Based on the above discussion, scaling at every stage and using current mirrors to do connection between building blocks is a good method for cicuit performance and silicon technology consideration. By implementing the scaling circuit and the current mirrors all in the building blocks. The building blocks have the general form shown in Figure 4.9. This kind of building block is used by Lustenberger [43][44][42] and Loeliger [38][40][39], and we call it the *canonical* design.

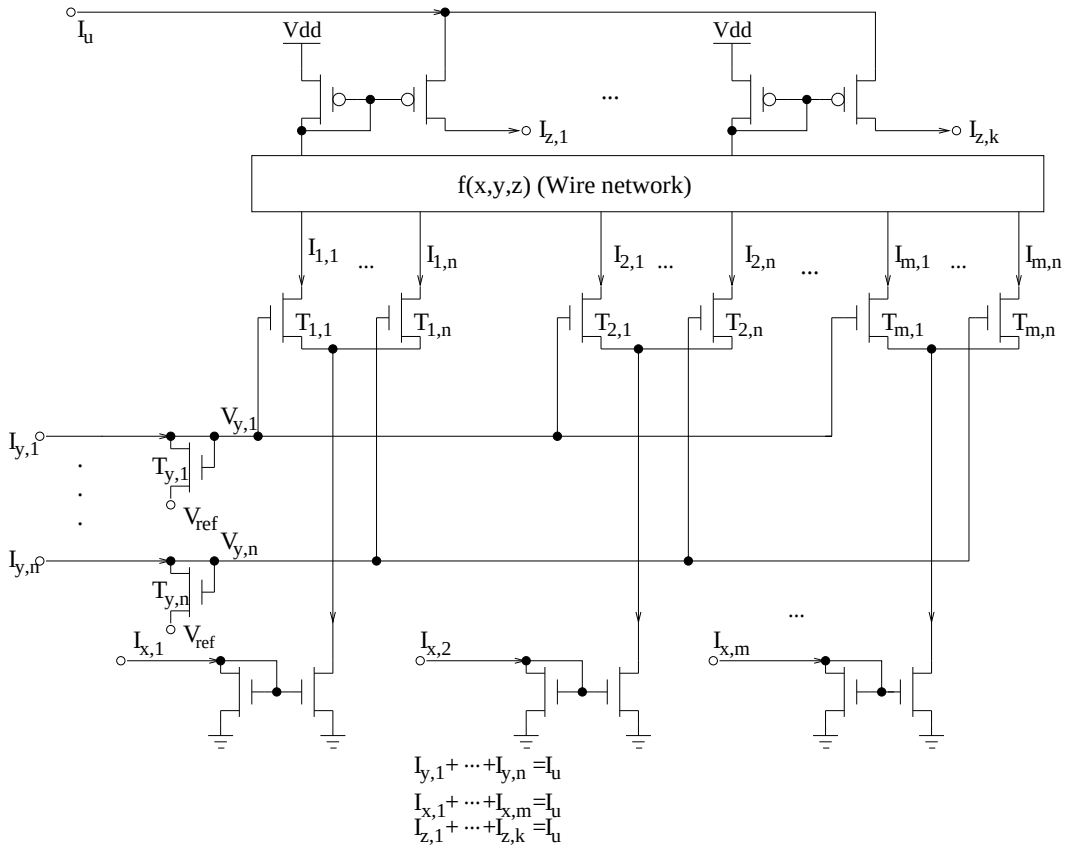

Figure 4.9. Typical cell used in canonical design.

4.4 Cell Library

From the general circuit for the building blocks shown in Figure 4.9, we know that there are a huge number of different building blocks depending on the different $f(x, y, z)$ shown in Figure 4.1. This makes automatic synthesis nearly impossible. This section shows that actually different building blocks can be built by using circuit cells from a cell library that has a limited number of circuit cells, making automatic synthesis practical.

4.4.1 Basic Cell

For the canonical basic building block shown in Figure 4.9, the values of m, n, k are allowed to vary. There are also many possible functions $f(x, y, z)$, making the number of such building blocks nearly infinite. Suppose that each z term is the sum of $l = xy/z$ product terms, there are

$$\begin{pmatrix} l \\ xy \end{pmatrix}\begin{pmatrix} l \\ xy-l \end{pmatrix}\begin{pmatrix} l \\ xy-2l \end{pmatrix}\cdots\begin{pmatrix} l \\ l \end{pmatrix}$$

different combinations, each requires a unique cell. For example, if $m = n = k = 4$, and each z term is the sum of 4 xy product terms, there are 63,063,000 different cells. Of course, we must find some small basic cells so that the number of basic cells can be kept under control.

In Figure 4.9, the circuit below the wire network is used to generate product terms and the circuit above the wire network is used to do normalization so that the current of all the outputs added together equal I_u where I_u is the current designated as representation of probability 1. Breaking the circuit shown in Figure 4.9 into two cells results in a product cell and a normalization cell as shown in Figure 4.10 and Figure 4.8, respectively.

In communication, the signal used always belongs to a signal set that have signal number of 2's power such as BPSK, QPSK, 8-PSK, and 16-QAM. Thus, m, n, k in Figure 4.1 are all powers of 2. Now, it seems that we could build a limited number of basic cells. However, if the output of a normalization cell needs to be distributed to the input of many product cells, then the current output of the normalization cell needs to be duplicated. Since the number of product cells to which the normalization cell's output

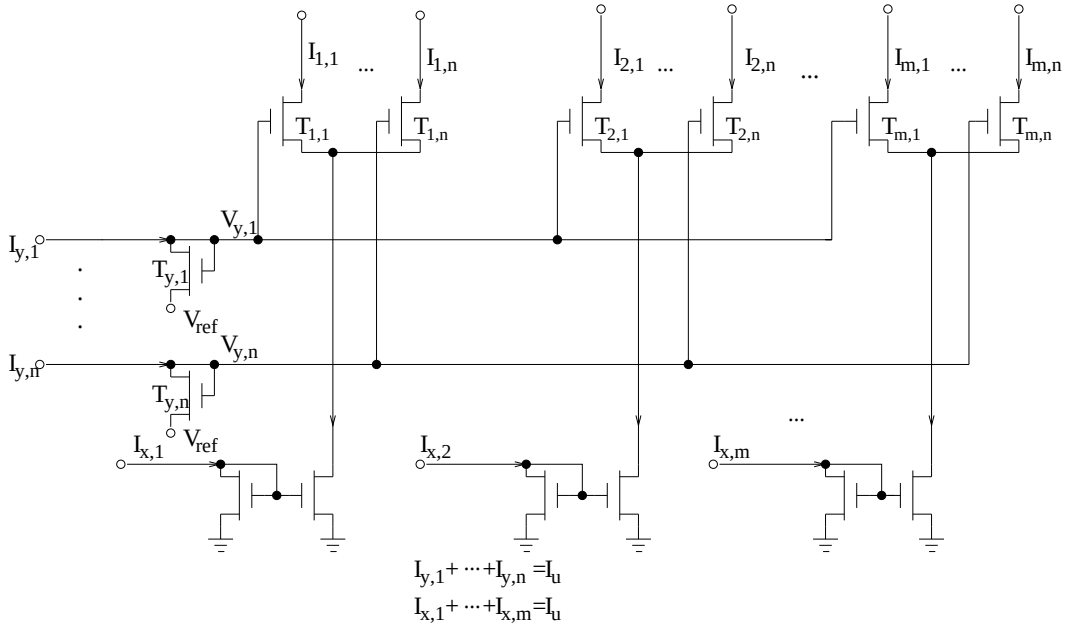

Figure 4.10. A current in, current out product cell.

must be distributed varies, there are a lot of different normalization cells if current mode connection is used between product cell and normalization cell. Therefore, it is better for us to move the diode connected transistors from the input of the product cell to the output of the normalization cell. As a result, the product cell accepts voltage input and provides current output where the normalization cell accepts current input and provides voltage output. Figure 4.11 shows a product-m-n cell and Figure 4.12 shows a normalization-k cell. Figure 4.13 shows the structure of building blocks and the connection of the building blocks using the proposed product cell and normalization cell. It is clear that half of the current mirror is in the normalization cell while the other half is in the product cell and thus voltage connection is used.

Note that V_{ref} in the normalization cell must be different based on whether its output is connected to the x input or y input of a product cell because the x input and y input of a product cell need different voltage potential. As a result, the output of a normalization cell must only be connected to either x inputs or y inputs of product cells. In the case when the output of a normalization cell needs to be connected to both the y inputs of product cells and x inputs of product cells, normalization cells that we call dnormalization-k cells shown in Figure 4.14 are needed.

From the above analysis, if $m \leq 2^r$, $n \leq 2^s$, and $k \leq 2^t$, then a cell library needs rs

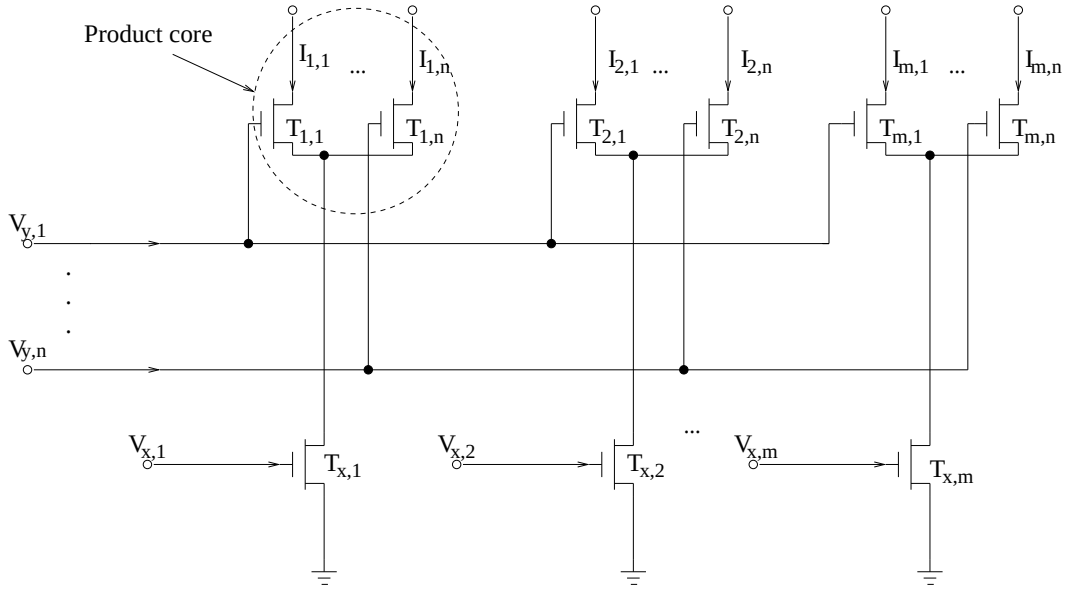

Figure 4.11. Product cell.

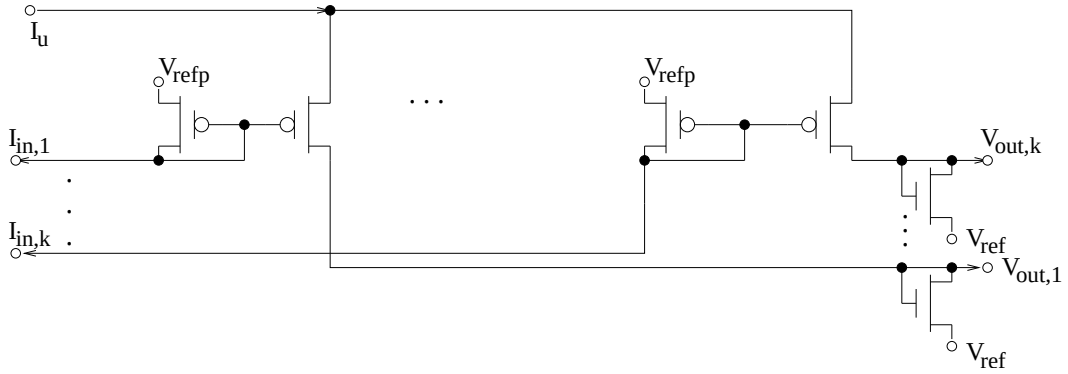

Figure 4.12. Normalization cell.

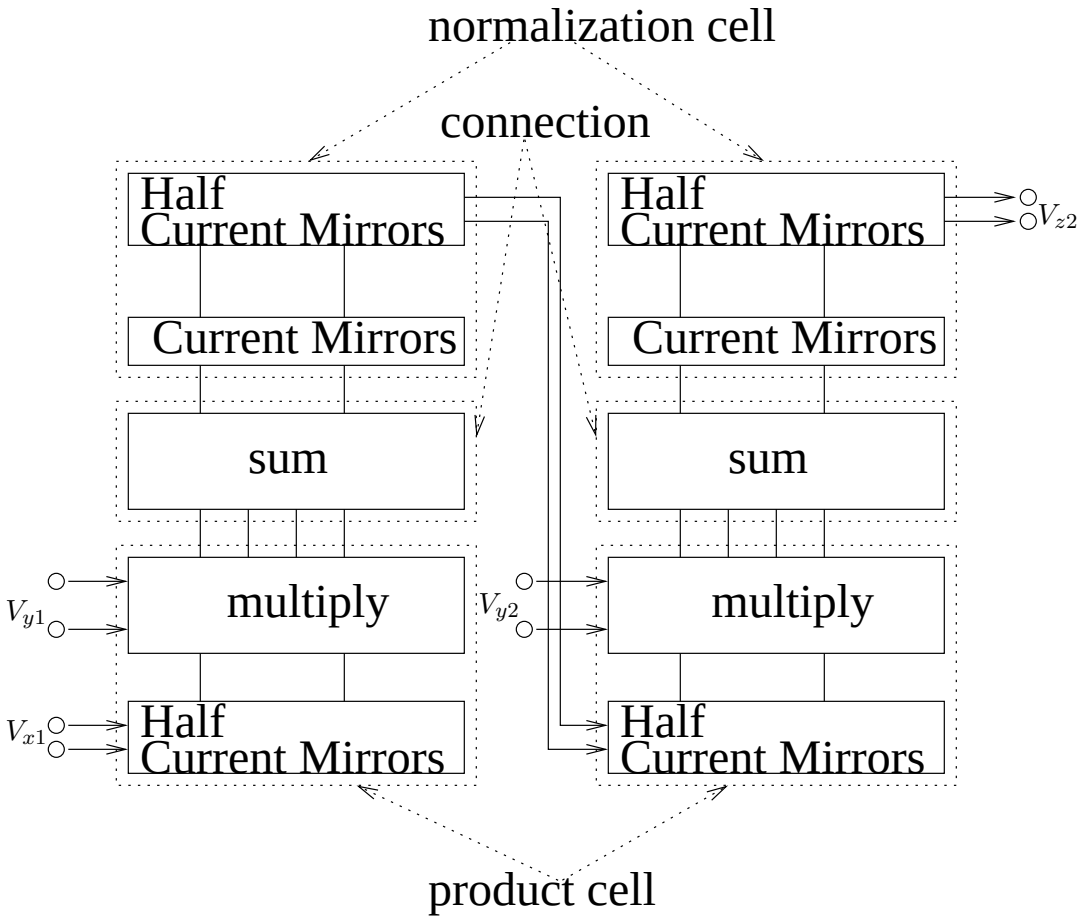

Figure 4.13. Building blocks using the proposed cell.

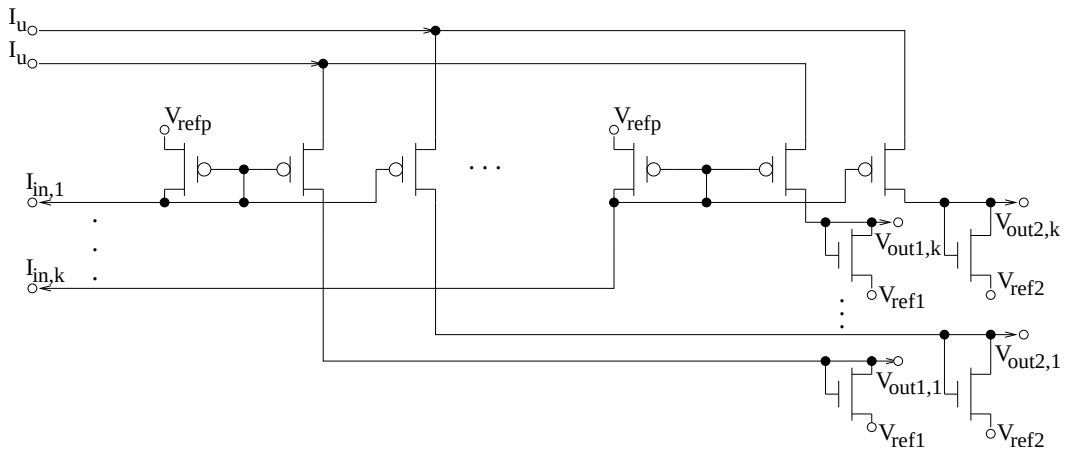

Figure 4.14. Double normalization cell.

product-m-n cells, t normalization-k cells and t dnormalization-k cells. Our current cell library has 80 cells: product-2-2, ..., product-2-256, ..., product-256-256, normalization-2, normalization-4, ..., normalization-256, dnormalization-2, dnormalization-4, ..., dnormalization-256, and it enables construction of current analog error control decoders. This is more than sufficient for current analog error control decoders in which typical values of m, n, k are not larger than 16. As a result, a cell library with 24 cells is enough for current analog error control decoders. Figure 4.15 and 4.16 are two examples of how to build larger blocks using these cells. The circuit shown in Figure 4.15 calculates the joint probabilities of two independent variables. The circuit shown in Figure 4.16 is the butterfly trellis (also called soft XOR). It performs the following operation and it is often used when Z is a parity check variable of X and Y.

$$\begin{bmatrix} pZ(0) \\ pZ(1) \end{bmatrix} = \begin{bmatrix} pX(0)pY(0) + pX(1)pY(1) \\ pX(0)pY(1) + pX(1)pY(0) \end{bmatrix}$$

4.4.2 Thermal effect

A potential problem of using these basic cells to build a circuit is cross-chip temperature difference. The voltage-current relation has dependence on temperature. As a result, using this approach, this thermal effect may be greater than the canonical design because the temperature between different blocks may be different. As a result, this thermal effect is analyzed by using the equations described in [65] and choosing typical values for $k_3 = 1.5$ and $k_4 = 2mV/K$. For the circuit shown in Figure 4.17, corresponding Matlab

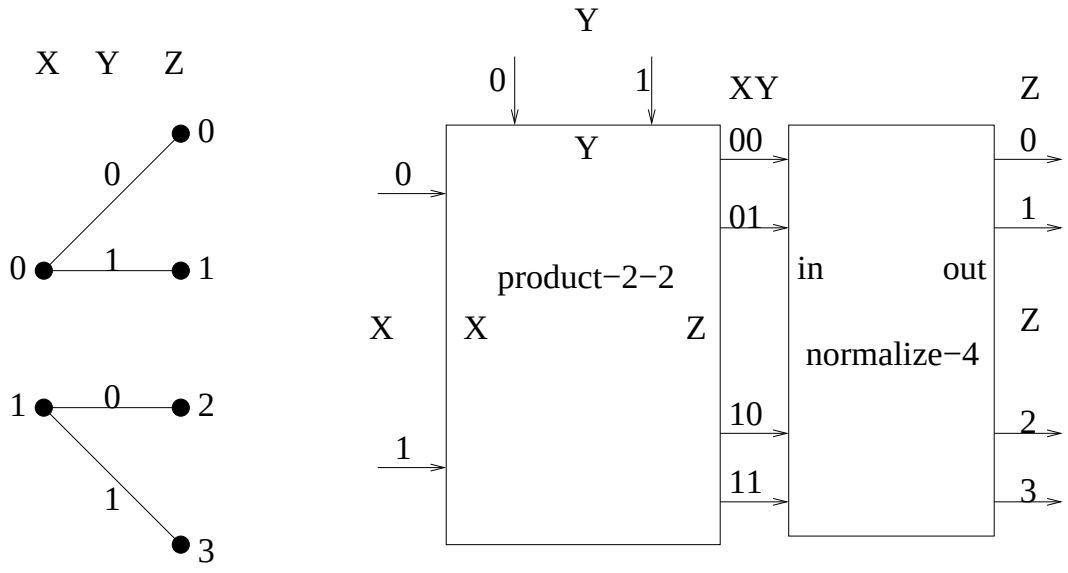

Figure 4.15. Example 1.

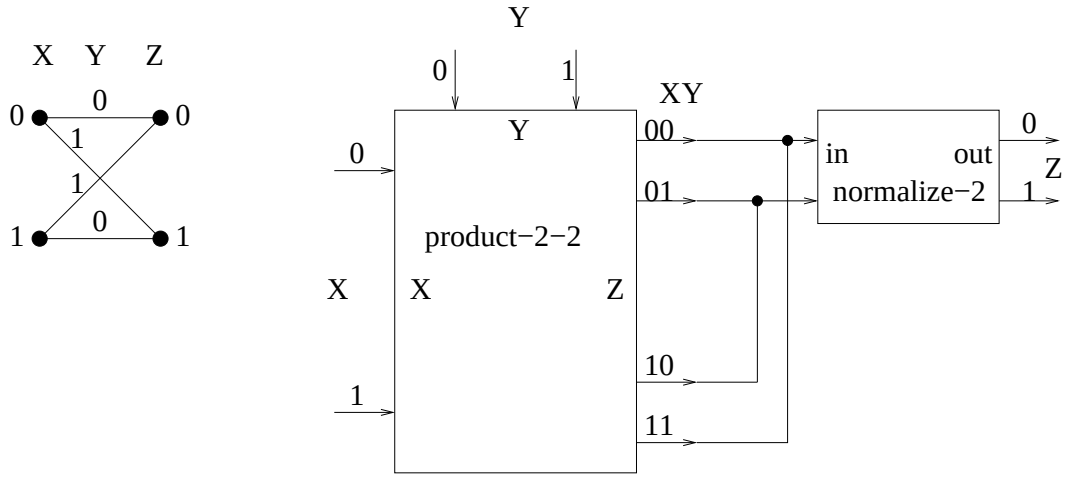

Figure 4.16. Example 2.

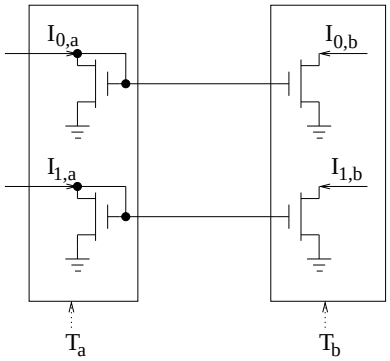

Figure 4.17. Current mirrors.

simulation shows that even if $I_{0,b}$ and $I_{1,b}$ are changed significantly compared with $I_{0,a}$ and $I_{1,a}$, the probability $p_{0,b} = I_{0,b}/(I_{0,b} + I_{1,b})$ does not change significantly compared with $p_{0,a} = I_{0,a}/(I_{0,a} + I_{1,a})$ because the currents are magnified by nearly the same scale. Figure 4.18 shows the relationship between $p_{0,b}$ and $p_{0,a}$ if $T_a = 300K$, $T_b = 310K$, and $I_{0,a} + I_{1,a} = 1nA$. It is clear that even $10K$ variation does not change the probability much. Figure 4.19 and Figure 4.20 shows the relationship between $p_{0,b}$ and $p_{0,a}$ when T_b is changed to $320K$ and $330K$ respectively. It shows that even $20K$ and $30K$ variation only change the probability slightly. For analog error control decoders, the circuit blocks are uniformly distributed in the chip so that the power consumed is also nearly uniformly distributed. Also, for a CMOS circuit working under weak inversion, the consumed power is very small. As a result, the temperature difference between different blocks should be small and thermal effect should not be a serious problem.

4.4.3 Decreasing the Circuit Complexity

Note from Figure 4.11 that in the product core, there are always n transistors even if only k products are useful and the other $n - k$ products are discarded by shorting their outputs to some voltage soure $Vdummy$. These dummy transistors are needed to make the following equation hold for the current shown in Figure 4.9.

$$I_{i,j} = I_{x,i} * I_{y,j}/(I_{y,1} + ... + I_{y,n}) = I_{x,i} * I_{y,j}/I_u$$

We can minimize the transistor count by substituting the $n - k$ transistors with 1 transistor, if the circuit shown in Figure 4.21 is used to generate the gate voltage of this transistor. Figure 4.21 contains $k + 2$ transistors (1 PMOS transistor and $k + 1$ NMOS

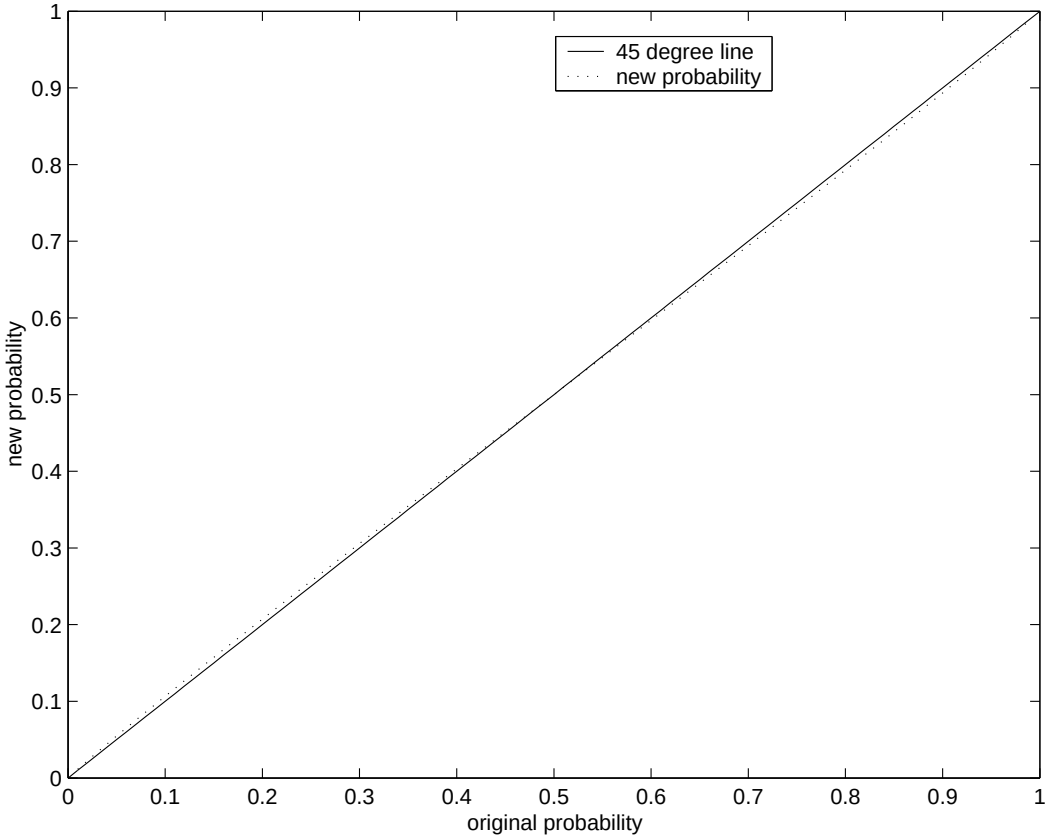

Figure 4.18. Thermal effect of 10K temprature difference for the current mirrors.

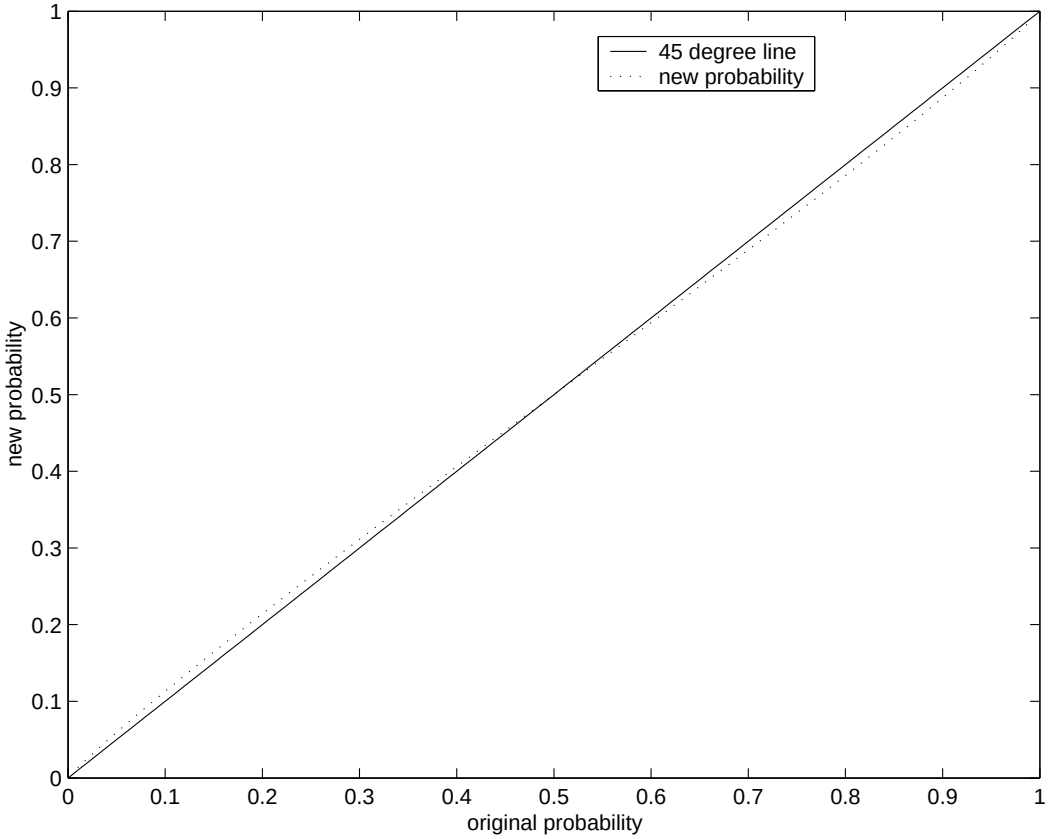

Figure 4.19. Thermal effect of 20K temprature difference for the current mirrors.

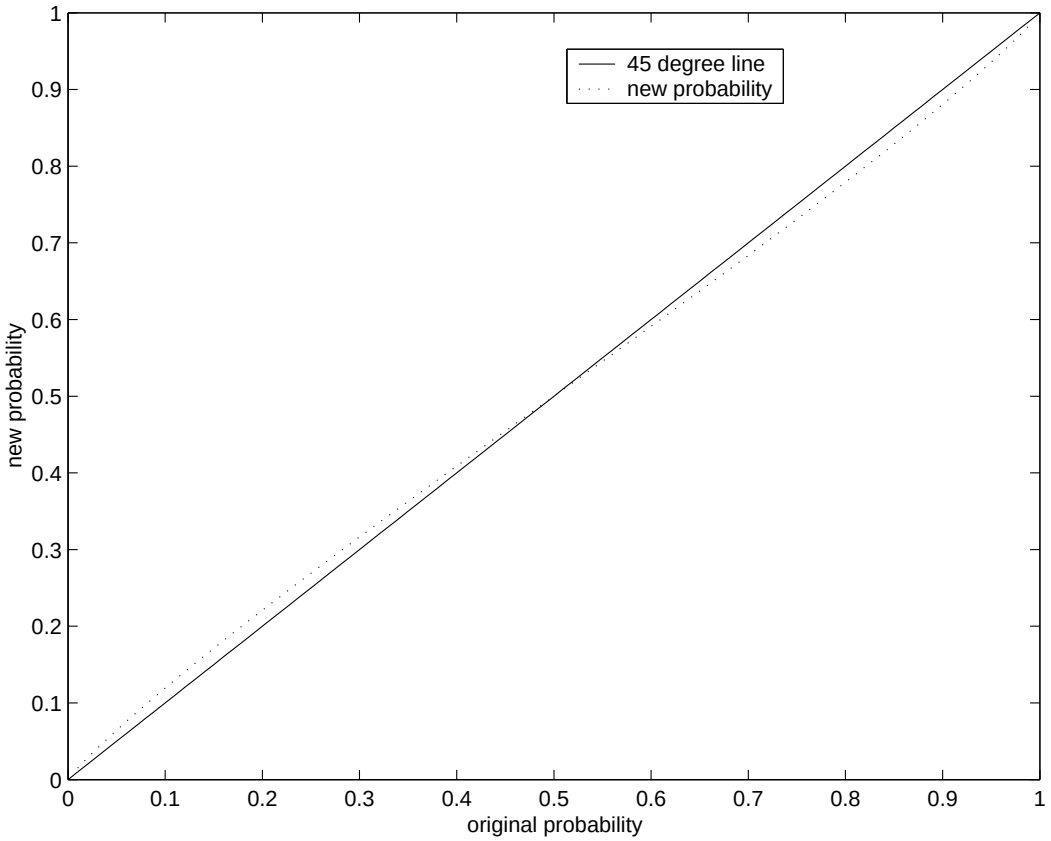

Figure 4.20. Thermal effect of 30K temprature difference for the current mirrors.

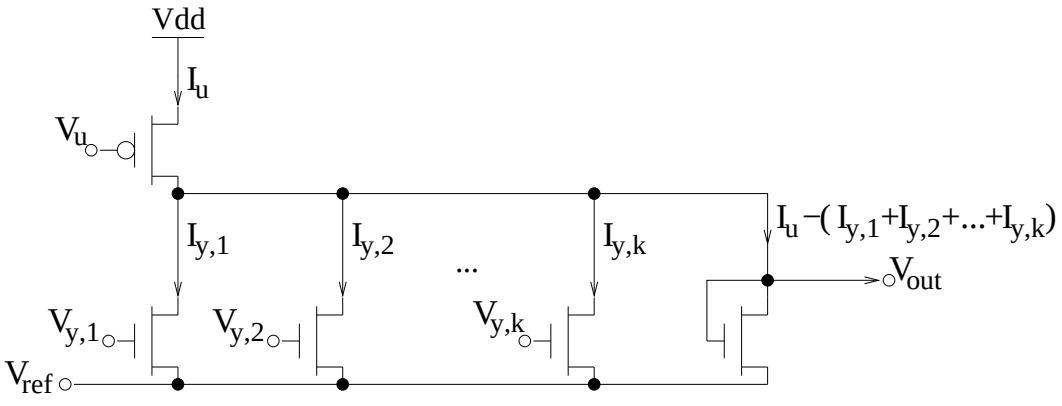

Figure 4.21. Reference cell.

transistors). As a result, we can replace the original $n-k$ transistors with $k+3$ transistors. In cases in which n is much larger than k, using this technique can save a large quantity of transistors. The disadvantage of using this technique is that the circuit may be a little slow compared with the original design. Also, notice that if this technique is used, voltages $V_{y,1}, V_{y,2}, ...V_{y,k}$ are provided by a normalization cell while the voltage V_{out} shown in Figure 4.21 is provided by a reference cell. These voltages are provided by different cells. However, they are provided as the input to a product cell. As a result, thermal effect may cause problems in this case.

In order to use this technique, we need to build another family of basic cells that we call reference cells. This kind of cell always contains $k+1$ NMOS transistors and 1 PMOS transistor. Since k is always a power of 2, building 7 cells ref-2, ref-3, ref-5, ..., ref-129 are enough for most analog error control decoders.[1] We also need to build corresponding product cells, product-2-3, product-2-5, ..., product-2-129.

4.4.4 Comparison with Canonical Design

The circuit cell shown in Figure 4.9 are used in canonical design. In Figure 4.9, the product cell and normalization cell are combined together, and they all use current input and current output. The advantages are that the thermal problem is reduced and the wires between the product cell and the normalization cell are short because the two cells are combined in one cell. However, the thermal problem is likely not a serious problem for our approach from the previous analysis. Also, the wire network does not significantly affect the performance. As a result, the circuit performance using our approach should still be comparable with the canonical design. We have verified the performance to be nearly the same with Spice simulation of the extended Hamming (8,4) decoder [69] using this approach and the canonical design.

For the circuit complexity, when the outputs of a cell shown in Figure 4.9 are needed to be given to several cell's input, then the output current needs to be duplicated by using more transistors. For the extended Hamming (8,4) decoder [69], the core of the decoder uses 292 transistors using this approach. If the canonical design shown in Figure 4.9 is used, then the core of the decoder requires 36 more transistors.

This approach is also lower power because current duplication is saved in some cases. For the extended Hamming (8,4) decoder [69], the power used by the core of the decoder

[1] We use the number of NMOS transistors in the cell to discriminate different cells.

using this approach is $32 * I_u * Vdd$ while the power used by the corresponding canonical design is $40 * I_u * Vdd$ because eight current duplications are saved while each current duplication consumes $I_u * Vdd$.

4.4.5 The Other Two Choices

There are still two other choices in breaking the canonical block into product cell and normalization cell. In both of these two choices, the product cell and normalization cell shown in Figure 4.10 and Figure 4.8 that accept current input and provide current output are used. For the first choice, if the output of a normalization cell needs to be provided to m product cells. Then the current is mirrored by m current mirrors to provide the outputs needed. Fortunately, the number m is usually not large. For general cases, $m = 4$ is enough. As a result, normalization-2-1, normalization-2-2, ... normalization-2-4, normalization-4-1, ..., normalization-256-1, ..., normalization-256-4 need to be provided. Using this approach, there is no thermal effect problem. However, the number of normalization cells is doubled even for $m = 4$ compared with the cell library that we have just described. The disadvantage is that more transistors are used. Also, more power is consumed compared with the previous cell library, especially when m is large for many normalization cells.

Another solution is to provide only one set of currents by the normalization cell. If the output of the normalization needs to be provided to m product cells. Then all the m product cell's inputs are connected to the output of the normalization cell and the output current of the normalization cell is equally distributed to the m product cells so that the input probability distribution is still what is required. Of course, if the output of a normalization cell needs to be connected to both the y input of product cells and x input of product cells, normalization cells that can provide two sets of current outputs shown in Figure 4.22 are needed. Using this approach, there is no thermal effect problem and we still get the same benefit of transistor and power savings as the previous cell library. However, since the current output of the normalization cell is connected to the m product cell's input, the current input of the m product cells is small compared with the previous cell library, especially when m is large, degrading the speed.

All in all, because the thermal effect is quite small, especially for such low power designs, we think that the previous cell library is the best and we choose this kind of cell library to do automatic synthesis.

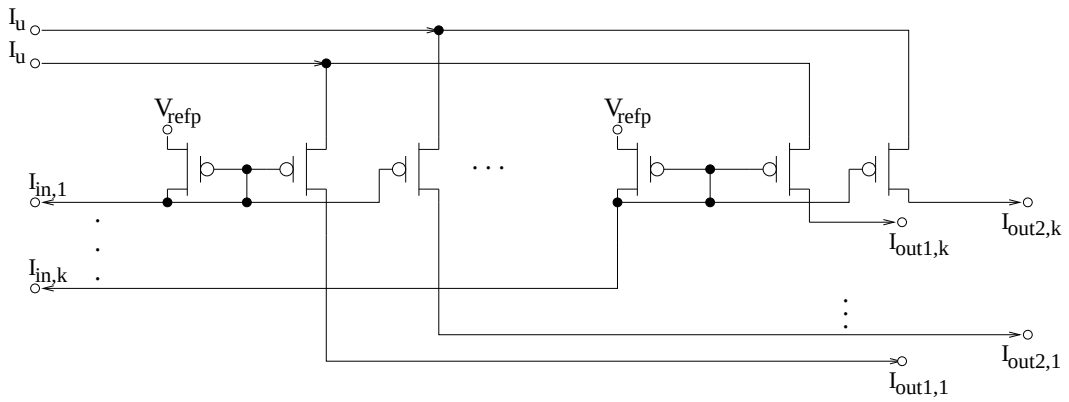

Figure 4.22. A normalization cell that provides two set of current output.

4.5 Circuit Structure

The previous sections discuss how to connect building blocks and how to construct the building blocks by using cells from a cell library. However, how to construct the building blocks to make a good structure has not been discussed. This section discusses this topic and also provides some novel circuits that can generate a better result in some cases.

4.5.1 Speed Consideration

Figure 4.23 is the typical core structure for an analog trellis decoder. The F_0, F_1, ..., F_{k-1} blocks construct the forward path of the trellis while the B_0, B_1, ..., B_{k-1} blocks construct the backward path of the trellis and the R_0, R_1, ..., R_{k-1} blocks are used to compute the result u_0, u_1, ..., u_{k-1}. For a tail-biting trellis, $\alpha_0 = \alpha_k$ and $\beta_0 = \beta_k$. This figure shows that during the computation of one code, γ_0, γ_1, ..., γ_{k-1} are stable while α_1, α_2, ..., α_k and β_0, β_1, ..., β_{k-1} change at least once. Looking at Figure 4.11, it is easy to see that the V_y-output current response is faster than the V_x-output current response. As a result, it is better to use γ_0, γ_1, ..., γ_{k-1} as the X input of a product cell to do computation along the forward and backward paths. For the R_0, R_1, ..., R_{k-1} blocks, it is better to calculate the product of $\alpha_{i-1}(i = 1, \cdots, k)$ and $\beta_i(i = 1, \cdots, k)$ first because the number of states is usually larger than the number of branches.

4.5.2 Complexity Consideration

The following four equations are needed for a MAP decoder.

$$P_r(s_{r-1} = i, s_r = j, y) = \alpha_{r-1}(i)\gamma_{r-1}(i, j)\beta_r(j) \tag{4.15}$$

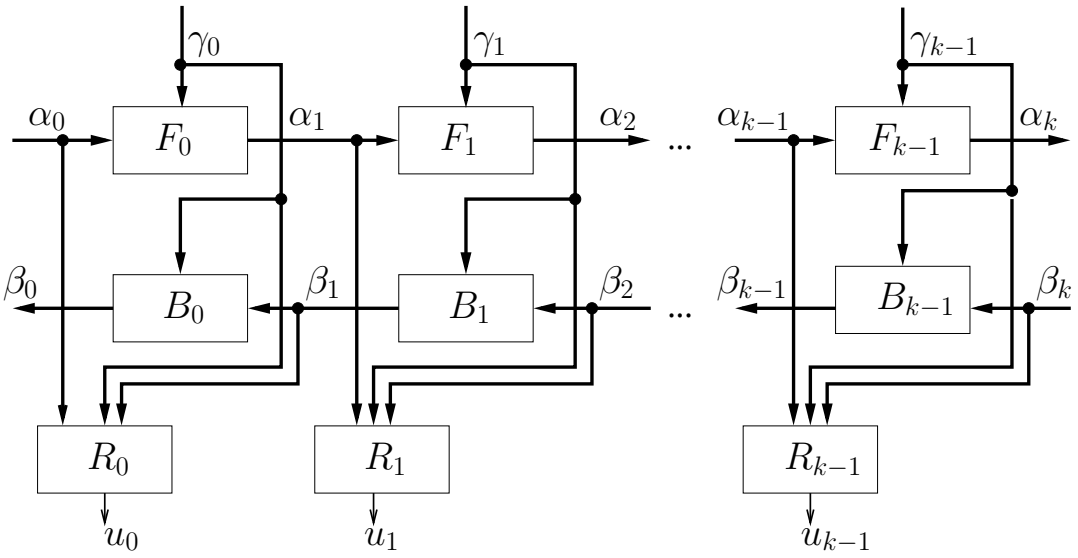

Figure 4.23. System structure for a typical trellis decoder.

$$\alpha_r(j) = \sum_{states\, i} \alpha_{r-1}(i)\gamma_{r-1}(i,j) \tag{4.16}$$

$$\beta_{r-1}(i) = \sum_{states\, j} \beta_r(j)\gamma_{r-1}(i,j) \tag{4.17}$$

$$P_{r-1}(u_{r-1} = u) = \sum_{(i,j)\in A(u)} P_{r-1}(s_{r-1} = i, s_r = j|y) \tag{4.18}$$

Since $P_{r-1}(s_{r-1} = i, s_r = j|y)$ can be reconstructed easily by dividing $P_{r-1}(s_{r-1} = i, s_r = j, y)$ by $P_{r-1}(y)$, which can be accomplished simply by normalizing Equation 4.15 to sum to unity. As a result, what is important is Equation 4.16, 4.17, and the following equation.

$$\begin{aligned} P_{r-1}(u_{r-1} = u) * P_{r-1}(y) &= \sum_{(i,j)\in A(u)} \alpha_{r-1}(i)\gamma_{r-1}(i,j)\beta_r(j) \\ &= \sum_{(i,j)\in A(u)} \gamma_{r-1}(i,j)\alpha_{r-1}(i)\beta_r(j) \end{aligned} \tag{4.19}$$

Let us assume that there are m states at time $r-1$, k states at time r and n different branches between time $r-1$ and time r and assume that the number of branches leaving

from a state at time $r-1$ is a and the number of branches entering a state at time r is b. It is apparent that ma=kb. Now, if we would like to calculate the α values at time $r-1$, β values at time r, and $\sum_{(i,j)\in A(u)} \gamma_{r-1}(i,j)\alpha_{r-1}(i)\beta_r(j)$, from the discussion above, we know that the complexity for calculating α values at time $r-1$ is $mn+n+3k$ ($mn+n$ transistors are used for the product cell and $3k$ transistors are used for the normalization cell). The complexity for calculating β values at time r is $nk+n+3m$. The complexity for calculating all possible combinations of $\alpha_{r-1}(i)\beta_r(j)$ is $mk+min(m,k)+3ma$. The complexity for calculating $\sum_{(i,j)\in A(u)} \gamma_{r-1}(i,j)\alpha_{r-1}(i)\beta_r(j)$ is $nma+n+3*2$ (assume u is binary). Usually the number of states is greater than the number of different branches. As a result, the complexity is mainly dominated by the product of the two numbers of states at adjacent times. If a is smaller than m and k, we can use the technique described in Section 4.4.3 so that the complexity for calculating all the possible combinations of $\alpha_{r-1}(i)\beta_r(j)$ is decreased to $min(m*min(2a+3,k)+m, k*min(2b+3,m)+k)+3ma$. As a result, the complexity is dominated by the product of the number of states with the number of branches.

If a γ value is always connected to a certain u value in the trellis section (We call this condition 1 for later reference), then Equation 4.19 can be simplified to Equation 4.20. As a result, the complexity of calculating $\sum_{(\gamma)} \alpha_{r-1}(i)\beta_r(j)$ is $min(m*min(2a+3,k)+m, k*min(2b+3,m)+k)+3n$ and the complexity of calculating $\sum_{\gamma\in A(u)} \gamma_{r-1}(i,j)\sum_{(\gamma)} \alpha_{r-1}(i)\beta_r(j)$ is $n*min(n,4)+n+3*2$. It is apparent that $n<ma$. Thus, the complexity is decreased. For example, Figure 4.24 shows the butterfly trellis, which is often used in many decoders. In order to get the result of u that is also the encoded output x, $\sum \beta\gamma$ can be calculated first as shown in Figure 4.25. The calculation of $\sum \beta\gamma$ needs a product-4-2 and a norm-2 cell and the calculation using $\sum \beta\gamma$ and α needs a product-4-2 and norm-2 cell. However, noticing that there is a 1 to 1 relation between γ and u, we can calculate $\sum \alpha\beta$ first and then calculate the result of u just as shown in Figure 4.26. Using this method, only a product-2-2 cell and a norm-2 cell are needed to calculate $\sum \alpha\beta$ and a product-2-2 and a norm-2 cell are needed to get the result of u, resulting in a decrease in circuit complexity.

$$\begin{aligned} P_r(u_r=u)*P_r(y) &= \sum_{(i,j)\in A(u)} \alpha_{r-1}(i)\gamma_{r-1}(i,j)\beta_r(j) \\ &= \sum_{\gamma\in A(u)} \gamma_{r-1}(i,j)\sum_{(\gamma)} \alpha_{r-1}(i)\beta_r(j) \end{aligned} \tag{4.20}$$

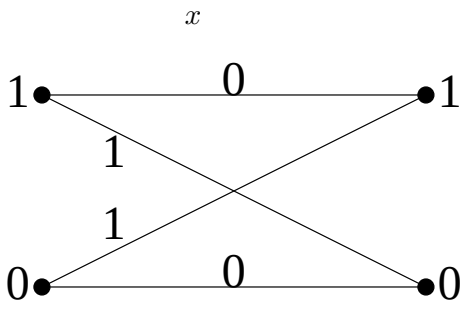

Figure 4.24. Butterfly trellis.

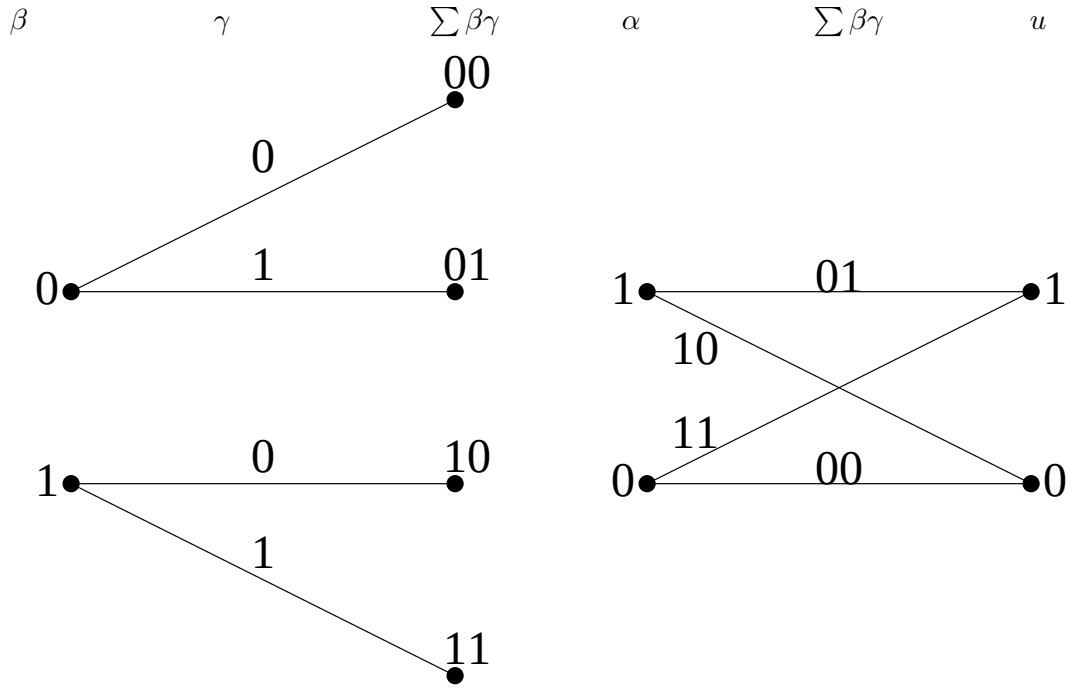

Figure 4.25. One decoding method.

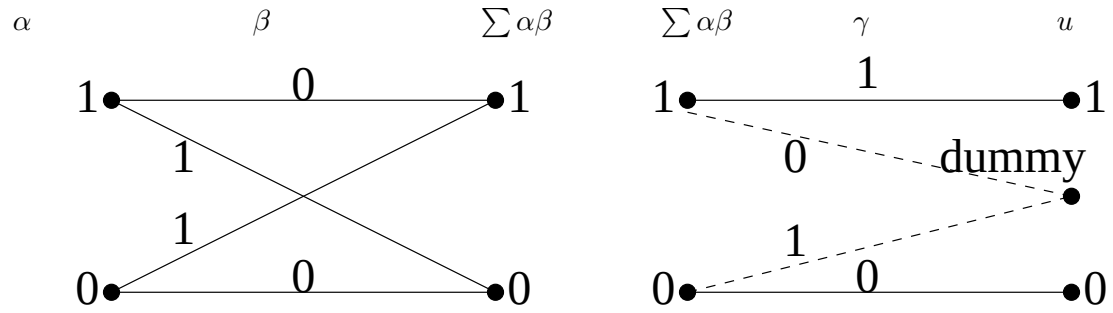

Figure 4.26. Another decoding method using the improved method shown in Equation 4.20.

If the branches that enter a state always belong to some $A(u)$ (We call this condition 2 for later reference), then Equation 4.19 could be simplified to Equation 4.21. As a result, the complexity for implementing such kind of code is decreased more because the complexity for calculating $\sum_{(j)\in A(u)} \alpha_r(j)\beta_r(j)$ is only $k * min(k,4) + k + 3 * 2$. For example, for the extended Hamming (8,4) code shown in Figure 2.11, the calculation of u_0 can be expressed by Equation 4.22.

$$\begin{aligned} P_r(u_r = u) * P_r(y) &= \sum_{(i,j)\in A(u)} \alpha_{r-1}(i)\gamma_{r-1}(i,j)\beta_r(j) \\ &= \sum_{(j)\in A(u)} \beta_r(j) \sum_i \alpha_{r-1}(i)\gamma_{r-1}(i,j) \\ &= \sum_{(j)\in A(u)} \alpha_r(j)\beta_r(j) \end{aligned} \tag{4.21}$$

$$\begin{aligned} p_{u_0}(0) &= \frac{p_{\alpha_0(0)}p_{\gamma_0(00)}p_{\beta_1(0)} + p_{\alpha_0(1)}p_{\gamma_0(01)}p_{\beta_1(2)}}{p_{\alpha_0(0)}p_{\gamma_0(00)}p_{\beta_1(0)} + p_{\alpha_0(1)}p_{\gamma_0(01)}p_{\beta_1(2)} + p_{\alpha_0(0)}p_{\gamma_0(11)}p_{\beta_1(1)} + p_{\alpha_0(1)}p_{\gamma_0(10)}p_{\beta_1(3)}} \\ &= \frac{p_{\alpha_1(0)}p_{\beta_1(0)} + p_{\alpha_1(2)}p_{\beta_1(2)}}{p_{\alpha_1(0)}p_{\beta_1(0)} + p_{\alpha_1(1)}p_{\beta_1(1)} + p_{\alpha_1(2)}p_{\beta_1(2)} + p_{\alpha_1(3)}p_{\beta_1(3)}} \\ p_{u_0}(1) &= \frac{p_{\alpha_0(0)}p_{\gamma_0(11)}p_{\beta_1(1)} + p_{\alpha_0(1)}p_{\gamma_0(10)}p_{\beta_1(3)}}{p_{\alpha_0(0)}p_{\gamma_0(00)}p_{\beta_1(0)} + p_{\alpha_0(1)}p_{\gamma_0(01)}p_{\beta_1(2)} + p_{\alpha_0(0)}p_{\gamma_0(11)}p_{\beta_1(1)} + p_{\alpha_0(1)}p_{\gamma_0(10)}p_{\beta_1(3)}} \\ &= \frac{p_{\alpha_1(1)}p_{\beta_1(1)} + p_{\alpha_1(3)}p_{\beta_1(3)}}{p_{\alpha_1(0)}p_{\beta_1(0)} + p_{\alpha_1(1)}p_{\beta_1(1)} + p_{\alpha_1(2)}p_{\beta_1(2)} + p_{\alpha_1(3)}p_{\beta_1(3)}} \end{aligned} \tag{4.22}$$

When doing automatic synthesis, we should check whether condition 1 or condition 2 exists. If condition 2 exists, we should use Equation 4.21. If condition 1 exists, we should use Equation 4.20. In other cases, we should use Equation 4.19.

4.5.3 Using Reset Circuits for Decoders with Cycles

Tail-biting trellis decoding is guaranteed to converge for the binary case [1][3]. In most case, the codeword is different from the previous codeword thus its trellis path also differs with the trellis path of the previous codeword. However, the initial condition for a codeword decoding is the end condition for the previous code word decoding for analog decoders because analog decoders work constantly. For conventional trellis decoders with no cycles as shown in Figure 4.23, $a(0)$ and $b(k)$ are always uniform ditributed probabilities and after k time units, all the probabilities along the forward path, $a(1), \ldots, a(k)$ and backward path, $b(0), \ldots, b(k-1)$ are decided only by the inputs $c(1), \ldots, c(k)$. Thus, the initial condition of $a(1), \ldots, a(k)$, $b(0), \ldots, b(k-1)$ does not have any affect to the decoding process. However, for decoders with cycles such as tail-biting trellis decoder in which $a(0) = a(k)$ and $b(0) = b(k)$, the probabilities along the cycles have feedback. As a result, the initial condition of the probabilities along the cycle can affect the decoding process through the feedback. This is very harmful, especially for high SNR applications, because the probabilities along the current codeword trellis path may be very small. In extreme cases, the probabilities along the current codeword trellis are so small that the decoder does not work. Let us use the tail-biting trellis of the extended (8,4) Hamming code shown in Figure 2.11 as an example. If there is no channel noise and the codeword is 0000000000, then after decoding, the decoder chooses a path represented by the state number 0-0-0-0 and the probability of these states are all 1. Then, if the next codeword is 10101010. The decoder cannot work because all the state probabilities along the path represented by the state number 1-3-1-3 have 0 probability. If we reset the probabilities of $a(1), a(2), ..., a(k)$ and $b(0), b(2), ..., b(k-1)$ shown in Figure 4.23 to unity distributions after one codeword is decoded, then the decoder could converge much faster. The simulation of the Hamming (8,4) decoder shown in Figure 4.27 shows that the speed can be nearly doubled for high SNR application ($SNR > 7$) if the reset circuit is used. In Figure 4.27, the flooding message passing schedule and a global synchronous clock is used. Four time units means that four clock cycles has passed since the arrival of the input probabilities. In many cases, simulation shows that the currents provided by the product cell are quite small such as $1.0^{-20}A$ when a new codeword arrives if no reset circuit is used. For the simulation, it is fine but for the real circuit, it may cause problems. Let us use the circuit shown in Figure 4.28 as an example (The currents connected to V_{dummy} are discarded and not used by the normalization cell.). Suppose that the channel

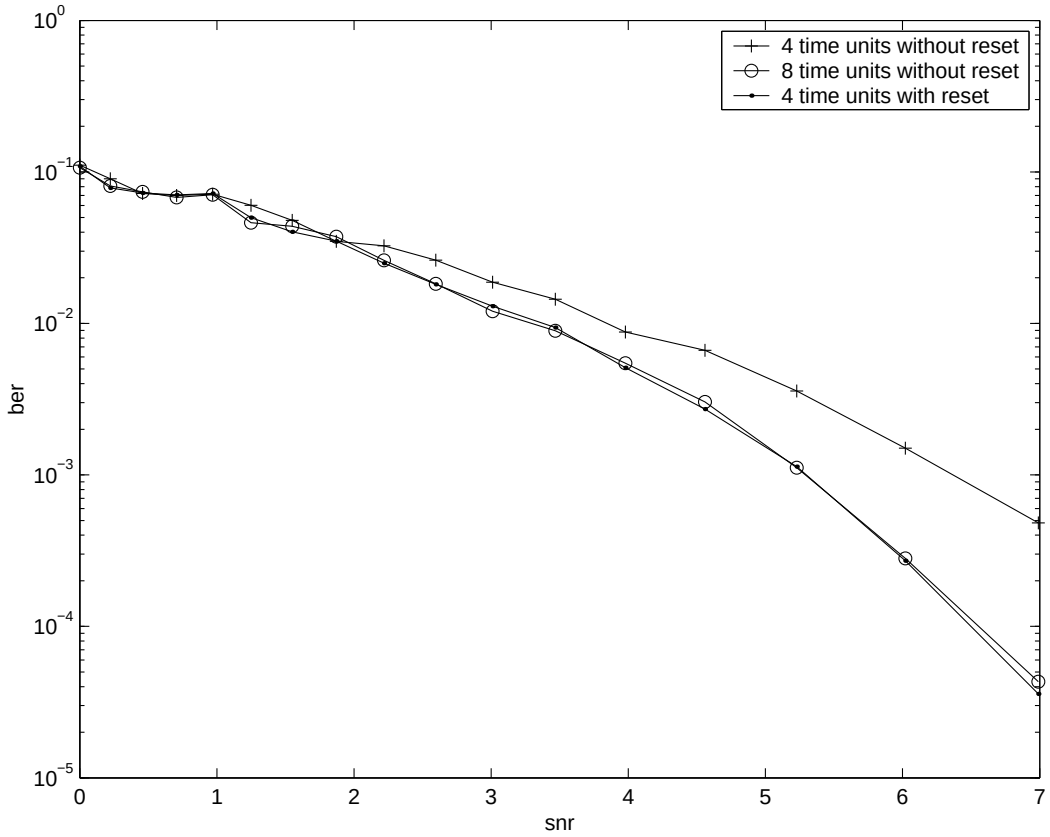

Figure 4.27. Simulation result of the extended Hamming(8,4) decoder using and not using a reset circuit.

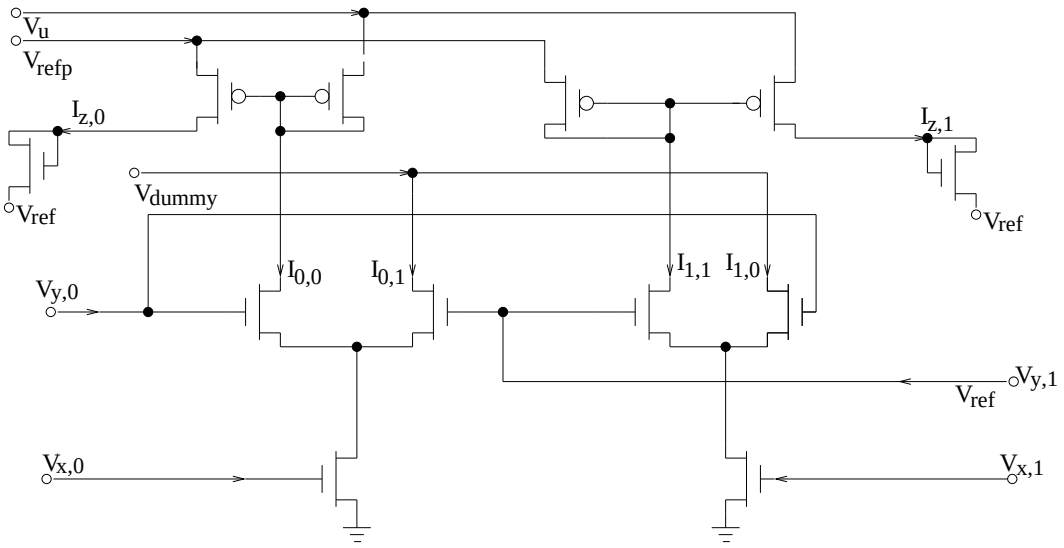

Figure 4.28. An example used to show the reset advantage.

information is provided as X input and Y input is in a cycle. If the previous codeword chooses the path of $V_{x,0}$, $V_{y,0}$, and $I_{0,0}$, then $V_{y,1}$ is small compared with $V_{y,0}$. Now, if for the current codeword, $V_{x,0}$ is small compared with $V_{x,1}$, then the useful currents $I_{0,0}$ and $I_{1,1}$ are both very small while the large current $I_{1,0}$ is discarded. The small currents make the circuit slower than expected. Also, if the useful currents provided by the product cell are all small, nonideal effects of the circuit causes more problems. As a result, a reset circuit is highly recommended, especially for high SNR applications.

For the reset circuit, we just add a transistor between the voltage outputs of the normalization cell as shown in Figure 4.29 so that when reset is high, the voltage outputs of the normalization cell are equal. Because V_{ref} is a low voltage and the circuit works in the subthreshold region, the output voltage of the normalization cell is a much lower voltage than Vdd. As a result, using only NMOS transistors to do the connection is enough.

However, using the reset circuit has the disadvantage that additional drain-bulk capacitance is added to the output of the normalization cell. As a result, the load capacitance of the normalization cell is increased, making the delay along the trellis path increased. Of course, we would like to add the smallest drain-bulk capacitances to the normalization cell. We would also like to make the load balanced for every output node of the normalization cell so that the worst case delay is not large. On the other

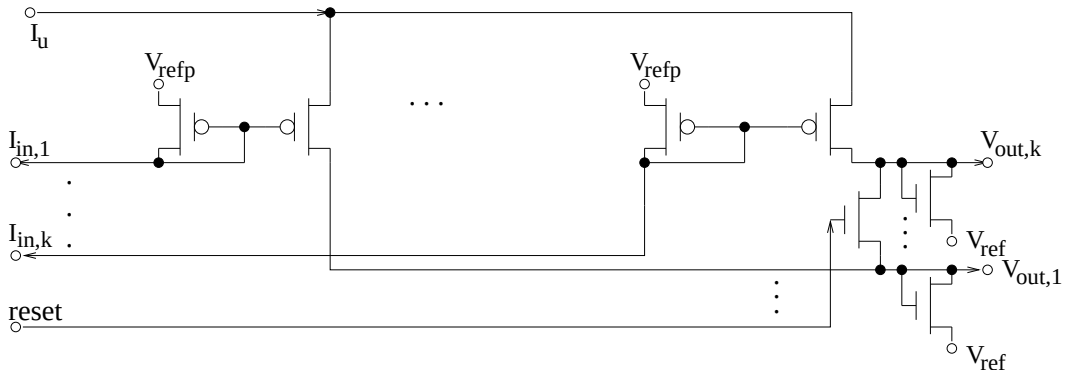

Figure 4.29. The normalization cell with reset control.

hand, we also need the resistances between the output nodes of the normalization cell when reset is high to be not large. As a result, the following connection scheme is used. For a normalization-k cell, $V_{out,1}$ is connected to $V_{out,2}$ by a NMOS transistor, $V_{out,2}$ is connected to $V_{out,3}$ by a NMOS transistor and so on until $V_{out,k-1}$ is connected to $V_{out,k}$ and $V_{out,k}$ is connected to $V_{out,1}$ to form a cycle. Also, notice that k is an even number, so a NMOS transistor is also used to connect $V_{out,l}$ and $V_{out,l+\frac{k}{2}}$ in which the value of l is between 1 and $\frac{k}{2}$ inclusive. Using this method, only $\frac{3k}{2}$ NMOS transistors are used to form the reset circuit for a normalization-k cell. The circuit is balanced. For every output node of the normaize-k cell, at most three additional drain-bulk capacitances are added to the load. Also, the resistance between any two output nodes is not large even for a large normalization-k cell when reset is high so that the circuit can be reset to generate unity distribution probabilities quickly.

Still, one needs to decide whether using this technique can improve the performance of the decoders with cycles or not. On the one hand, using a reset circuit can give a better initial condition for the circuit to start with, on the other hand, the additional drain-bulk capacitance adds additional delay to the circuit.

4.5.4 Power Consumption

For trellis codes, we know that the circuit used to realize Equation 4.19 is the most complex one. It uses a large number of transistors and consumes much power. However, we only need to use Equation 4.19 to find the final result when the probabilities along the trellis have stabilized. This means that we do not need the circuit realizing Equation 4.19

to be active all the time to consume power. As a result, we can make this part of the circuit inactive most of the time. This can be implemented by a simple circuit to control the power provided to the normalization cells of this part of the circuit as shown in Figure 4.30. Using this circuit, when control is low, there is no power provided to the normalization cell and power is saved.

4.6 From Factor Graphs to Basic Cells

The previous sections describe a cell library and how an analog decoder is built using the cells specified in the cell library. Also, Chapter 3 shows that every decoder can be described by its normal graph description. If every function node of the normal graph is implemented by some circuit block, then by connecting all blocks according to the normal graph connection, a circuit can be built. For analog implementation of a decoder, a function node can always be implemented by basic building blocks shown in Figure 4.1 while each basic building block can be implemented by a product cell and a normalization cell described in the cell library. Now, what we need to discuss is how to partition a function node into basic building blocks and how to partition the basic building blocks

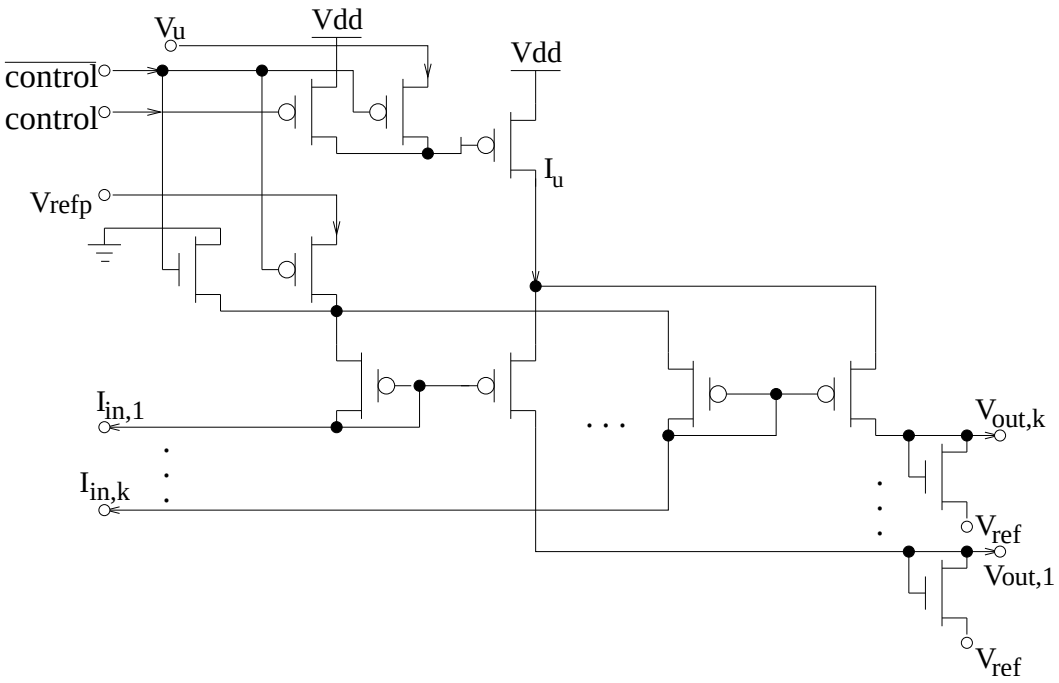

Figure 4.30. The normalization cell with power control.

into cells.

There are two kind of analog decoders. One kind uses a trellis to do decode and the other kind directly uses the parity-check functions. For trellis decoders, one section of the trellis is shown in Figure 3.9. The channel information is provided by a probability density function and for binary applications it can be implemented by the circuit shown in Figure 4.31 in which the drain-source current represents the probability provided by the channel information and the voltage input $\delta V = \frac{U_T}{\kappa}\frac{-4y}{N_0}$ (This can be verified by using Equation 2.25 and Equation 4.4.). For the function describing the trellis section, there are four variable nodes s_i, s_{i+1}, x_i, u_i. However, u_i is a leaf node so that when we calculate the α and β values, we are only concerned with s_i, s_{i+1}, x_i as shown in Equation 3.21 and Equation 3.22. As a result, only one basic building block is needed to implement Equation 3.21 and Equation 3.22 each. As described in the previous section, the γ messages shown in Figure 3.9 should be provided as X direction input for a building block shown in Figure 4.9 for speed consideration. Let us assume that in Figure 3.9, s_i has m states, s_{i+1} has k states, and x_i has n states. Then a product-n-m cell and a normalization-k cell are needed to implement Equation 3.21 and a product-n-k cell and a normalization-m cell are needed to implement Equation 3.22. For the implementation of Equation 3.21 and Equation 3.22, the normalization cell should generate outputs that can be connected to the Y direction input of a product cell so that they can be used for the next stage. For the implementation of Equation 3.23, if Equation 4.21 can be used, then only one building block is needed and it is composed of a product-n-n cell and a normalization-2 cell. In other cases, all the product items of two of α, β, γ need to be generated first before the result can be calculated. If Equation 4.20 can be used, then all the products of $\alpha_{r-1}(i)\beta_r(j)$ need to be generated first. When the product of $\alpha\beta$ is needed, we need to provide one of them as the X direction input of a product cell. As a result, in this case, instead of using normalization cells to generate the output, dnormalization cells need to be used for the implementation of either Equation 3.21 or Equation 3.22.

For decoders that are implemented directly on the parity-check equations as shown in Figure 3.4. There are only two building blocks needed, the equal gate and XOR gate. (The variable nodes with degree larger than 2 can be regarded as equal gates.) The circuit for an equal gate using two inputs to generate one output is shown in Figure 4.32. (The currents connected to V_{dummy} are discarded and not used by the normalization cell.) The

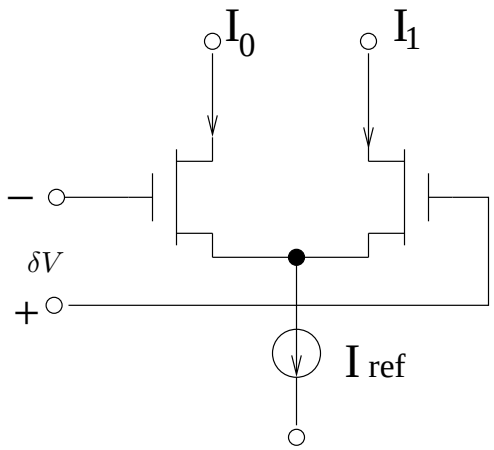

Figure 4.31. Transistor level implementation of conditional probability distribution based on channel information.

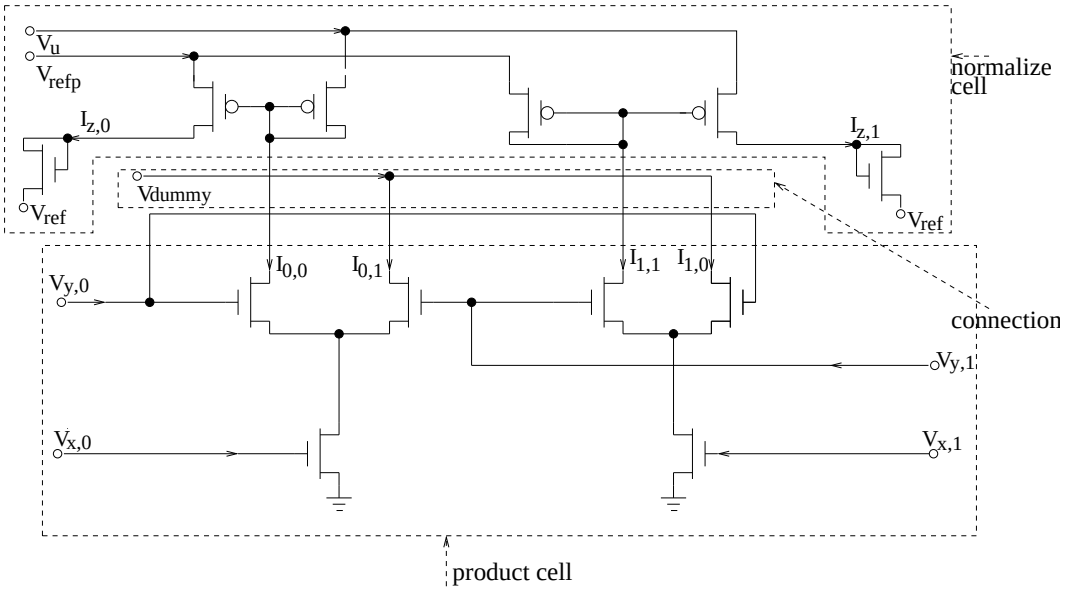

Figure 4.32. Transistor level implementation of the building block of an equal gate.

circuit for an XOR gate using two inputs to generate one output is shown in Figure 4.33. To generate one output of an equal gate function node with n connections, $n-2$ circuits shown in Figure 4.32 need to be used because $n-1$ inputs are used to generate the output. The first instance of the circuit shown in Figure 4.32 uses two inputs while additional instances of the circuit shown in Figure 4.32 use one of the remaining $n-3$ inputs each time. As a result, $k(n-2)$ instances of the circuit shown in Figure 4.32 are needed to implement an equal gate function node that needs to provide outputs to k of its adjacent nodes. A similar analysis shows that for an XOR gate function node with connections to n variable nodes, $k(n-2)$ instances of the circuit shown in Figure 4.33 are needed to implement an XOR gate function node that needs to provide outputs to k of its adjacent variable nodes.

For iterative decoding, instead of only using the channel information as the γ message shown in Figure 3.9, we need to combine the extrinsic probabilities provided by other component decoders with the channel information and provide the result as the γ information. Also, the decoder result should be generated by using all the extrinsic probabilities and the channel information. All these things are implemented by an equal gate as shown in Figure 3.4 (Actually, directly using parity-check equations to decode is a kind of iterative decoding) while the implementation of an equal gate is provided above.

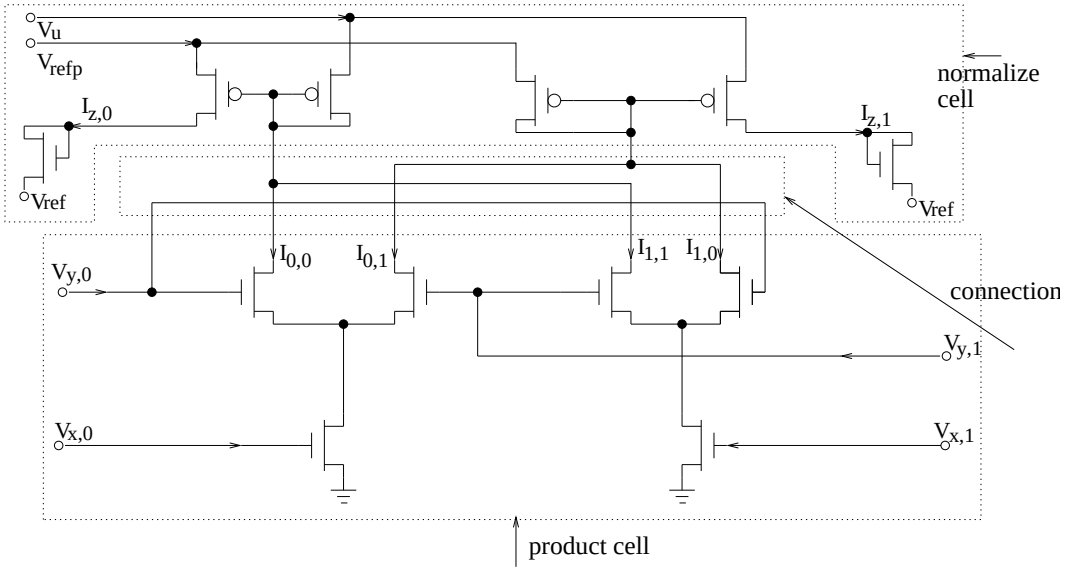

Figure 4.33. Transistor level implementation of the building block of an XOR gate.

Also, we can use other techniques to improve the performance of decoders with cycles and lower the power consumption. Using these techniques, the structure of the circuit are the same while different normalization cells are used.

Using the method mentioned above, we can convert the VHDL behavioral description of a function node to a VHDL structural description by describing how a function node is constructed by using the cells in the cell library. The previous chapter described that from a simple factor graph description of a decoder, the automatic tool can generate the VHDL behavioral description of a function node. As a result, an automatic synthesis tool is built. From the factor graph description of a decoder, the tool can generate the VHDL structural description of its analog implementation. Also, by using a commercial tool such as Silicon Ensemble, the schematic and layout of the analog decoder can be automatically generated. Thus the design process of an analog decoder is greatly sped up. The automatically generated circuit for some decoders and their performances are shown in Chapter 6.

CHAPTER 5

CIRCUIT LEVEL MODELING AND SIMULATION

Chapter 3 describes the behavioral level simulation of a decoder from its factor graph description. Chapter 4 describes how to implement decoders using analog circuits. For analog implementations, nonideal effects can affect the performance. In principal, all such nonideal effects can be studied by SPICE-level Monte-Carlo simulations. However, just as described in Chapter 3, SPICE level simulation is too time consuming to get the bit error rate curve and is even more time consuming for nonideal effects simulation. This chapter describes the circuit level modeling and simulation issues including all the major nonideal effects. Simple methods to model these nonidealities are provided and techniques to minimize these nonideal effects are developed. Since a large number of the nonideal effects can be represented by variables with Gaussian distribution, we first discuss the operations on variables with Gaussian distribution to give the basis for the following description.

5.1 Operations on Variables with Gaussian Distribution

If X is a random variable with Gaussian distribution and its mean is zero and its standard deviation is σ, then its probability density function is expressed by the following function.

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}} \quad (-\infty < x < \infty) \tag{5.1}$$

If Y is the linear combination of n independent Gaussian distribution variables $X_i, (i = 1, \ldots, n)$ all with mean 0 and standard deviation σ_{x_i}, i.e. $Y = a_1X_1 + a_2X_2 + \cdots + a_nX_n$, it is proven that the following equation is true.

$$standard deviation\, of\, Y\, is\, \sigma_y = \sqrt{a_1^2\sigma_{x_1}^2 + a_2^2\sigma_{x_2}^2 + \cdots + a_n^2\sigma_{x_n}^2} \tag{5.2}$$

If $\sigma_{x_1} = \sigma_{x_2} = \cdots = \sigma_{x_n} = \sigma$, then Y is a Gaussian distribution variable with mean 0 and standard deviation $\sqrt{a_1^2 + a_2^2 + \cdots + a_n^2}\sigma$ and we can write Equation 5.3 to mean that Y is a Gaussian distribution variable with mean 0 and standard deviation that is $\sqrt{a_1^2 + a_2^2 + \cdots + a_n^2}$ times of X_1's standard deviation for simplicity.

$$\begin{aligned} Y &= a_1X_1 + a_2X_2 + \cdots + a_nX_n \\ &= \sqrt{a_1^2 + a_2^2 + \cdots + a_n^2}X_1 \end{aligned} \tag{5.3}$$

Also, if X is a Gaussian distribution variable with mean 0 and standard deviation σ, then $1 + X$ and $1 - X$ are all Gaussian distribution variables with mean 1 and standard deviation σ. Also, if $Y = \frac{1}{1-X}$, then the probability density function for variable Y is shown in Equation 5.4.

$$f_Y(y) = (1 + \frac{1}{y^2})\frac{1}{\sqrt{2\pi}\sigma}e^{-\frac{(1-\frac{1}{y})^2}{2\sigma^2}} \quad (-\infty < y < \infty \; y \neq 0) \tag{5.4}$$

From Equation 5.4, we can see that Y is approximately a Gaussian distribution variable with mean 1 and standard deviation σ if $\sigma \ll 1$. As a result, we have the approximations shown in Equation 5.5 and Equation 5.6. These two equations are used in both the forward direction and backward direction later.

$$\begin{aligned} Y &= \frac{1}{1-X} \\ &\approx 1 + X \; (\sigma \ll 1) \end{aligned} \tag{5.5}$$

$$\begin{aligned} Y &= \frac{1}{a - bX} \\ &= \frac{1}{a}\frac{1}{1 - \frac{b}{a}X} \\ &\approx \frac{1}{a}(1 + \frac{b}{a}X) \; (a \gg b\sigma) \\ &= \frac{a + bX}{a^2} \end{aligned} \tag{5.6}$$

If X and Y are both Gaussian distribution variables with mean 0 and standard deviation σ and $Z = XY$, then the the probability density function of variable Z is shown in Equation 5.7.

$$f_Z(z) = \frac{1}{\sqrt{2\pi}\sigma}\int_{-\infty}^{\infty}\frac{1}{|y|}e^{-\frac{(\frac{z}{y})^2 + y^2}{2\sigma^2}}\,\mathrm{d}y \quad (-\infty < z < \infty) \tag{5.7}$$

From Equation 5.7, we can see that Z is approximately a Gaussian distribution variable with mean 0 and standard deviation $\sigma_z \approx \sigma_x\sigma_y = \sigma^2 \ll \sigma$ if $\sigma \ll 1$. Even if the

standard deviation of X, σ_x, and the standard deviation of Y, σ_y, are not equivalent, we can have $Z = XY = X(aY_1) = aXY_1$ in which a is a constant and the standard deviation of Y_1, $\sigma_{y_1} = \sigma_x$. Thus, the standard deviation of Z is $\sigma_z \approx a\sigma_x^2 = \sigma_x\sigma_y$. As a result, the approximations shown in Equation 5.8 and Equation 5.9 are true as long as the standard deviation of X and Y, $\sigma_x, \sigma_y \ll 1$.

$$Z = X(1+Y) = X + XY \approx X \ (\sigma \ll 1) \tag{5.8}$$

$$Z = (1+X)(1+Y) = 1 + X + Y + XY \approx 1 + X + Y = 1 + \sqrt{2}X \ (\sigma \ll 1) \tag{5.9}$$

Also, using Taylor expansion, Equation 5.10 and Equation 5.11 are true.

$$e^X = 1 + X + \frac{X^2}{2!} + \cdots + \frac{X^n}{n!} \tag{5.10}$$

$$\ln(1+X) = X - \frac{X^2}{2} + \frac{X^3}{3} - \frac{X^4}{4}\cdots \tag{5.11}$$

From the previous approximations, we have the following approximations.

$$e^X = 1 + X + \frac{X^2}{2!} + \cdots + \frac{X^n}{n!} \approx 1 + X \ (\sigma \ll 1) \tag{5.12}$$

$$\ln(1+X) = X - \frac{X^2}{2} + \frac{X^3}{3} - \frac{X^4}{4}\cdots \approx X \ (\sigma \ll 1) \tag{5.13}$$

5.2 Mismatch

The parameters of two identically designed devices on an integrated circuit show a random variation after fabrication, which is called *device mismatch*. Due to device mismatch, the fabricated circuit may not perform as designed. In general, device mismatch has much more impact on analog circuits than digital circuits. Several researchers have investigated the transistor mismatch effect [36] [55] [24] [53] [32] [5]. In general, transistor mismatch can be approximated by the mismatch in μ_n(NMOS), μ_p(PMOS), C'_{ox}, W, L, V_{T0n}(NMOS), and V_{T0p}(PMOS), each of which can be approximated by multiplying the designed value with $(1+\epsilon)$ in which ϵ is a gaussian distribution variable with zero mean and a small standard deviation. For example, the width of the transistor can be approximated by $W' = W(1+\epsilon_w)$. Also, all these mismatch effects are inversely proportional to the transistor area and this is one reason for using large transistors in an analog circuit.

By using Equation 4.4, Equation 4.5, and the equations from the previous section, we can approximate the drain-source current due to transistor mismatch also by multiplying the designed value with $(1+\epsilon)$ in which ϵ is a gaussian distribution variable with zero mean and a small standard deviation and this is shown in the derivation of Equation 5.14 using an NMOS transistor as an example. From Equation 4.4 and Equation 4.5, we know that drain-source current depend on gate-source voltage exponentially and U_T is small compared with κV_{T0n} so that the mismatch effect can be more serious for those analog circuits working in the weak inversion region than those working in the strong inversion region. Suppose that $V_{T0n} = 0.7V$, $\kappa = 0.7$ and $U_T = 26mV$, then a 0.4 percent mismatch (standard deviation $\sigma_{v_{t0n}}$ of the gaussian distribution variable $\epsilon_{v_{t0n}}$ shown in Equation 5.14) of V_{T0n} can generate a 7.5 percent mismatch (standard deviation σ of the gaussian distribution variable ϵ shown in Equation 5.14) in the drain-source current. Table 5.1 shows the drain-source current mismatch of NMOS transistors using MOSIS $0.5um$ technology and transistor width and length $3um$ [73]. Of course, transistor mismatch is enough to cause problems, especially when the transistors are working in the weak inversion region.

$$
\begin{aligned}
I_{DS} &= \frac{2\mu_n C'_{ox} U_T^2}{\kappa} e^{\frac{-\kappa V_{T0n}}{U_T}} \frac{W}{L} e^{\frac{\kappa V_G - V_S}{U_T}} \text{ (Using Equation 4.4)} \\
I'_{DS} &= I_{DS}(1+\epsilon_{\mu_n})(1+\epsilon_{ox})\frac{1+\epsilon_w}{1+\epsilon_l} e^{\frac{-\kappa V_{T0n}\epsilon_{v_{t0n}}}{U_T}} \text{ (Considering mismatch)} \\
&\approx I_{DS}(1+\epsilon_{\mu_n})(1+\epsilon_{ox})(1+\epsilon_w)(1-\epsilon_l) e^{\frac{-\kappa V_{T0n}\epsilon_{v_{t0n}}}{U_T}} \text{ (Using Equation 5.5)} \\
&\approx I_{DS}(1+\epsilon_{\mu_n})(1+\epsilon_{ox})(1+\epsilon_w)(1-\epsilon_l)(1-\frac{\kappa V_{T0n}}{U_T}\epsilon_{v_{t0n}}) \text{ (Using Equation 5.12)} \\
&\approx I_{DS}(1+\epsilon_{\mu_n}+\epsilon_{ox}+\epsilon_w-\epsilon_l-\frac{\kappa V_{T0n}}{U_T}\epsilon_{v_{t0n}}) \text{ (Using Equation 5.9 several times)} \\
&\approx I_{DS}(1+\epsilon) \text{ (Using Equation 5.2)} \\
\sigma &= \sqrt{\sigma_{\mu_n}^2+\sigma_{ox}^2+\sigma_w^2+\sigma_l^2+(\frac{\kappa V_{T0n}}{U_T})^2\sigma_{v_{t0n}}^2} \text{ (σ is the standard deviation of ϵ) (5.14)}
\end{aligned}
$$

As a result, Lustenbeger investigated the effect of mismatch on an analog decoder's performance [42]. Using some analysis, they obtained a high-level model for the basic cell of their analog decoder. Then, using the high-level model, they did the simulation for a

Table 5.1. Drain-source current mismatch.

Desired drain-source current	10nA	100nA	400nA	1uA	10uA
Drain-source current mismatch	9.37%	7.37%	5.49%	4.14%	1.78%

(44,22,8) low-density parity-check code. The simulation result shows that the performance degradation due to 10 percent drain-source current mismatch is not significant.

For the circuit shown in Figure 4.9 that is the basic building block for an analog decoder, we can analyze first the product part that is below the wire network because the analysis of the normalization part that is above the wire network is similar. For the product part, we analyze the Y input first. For simplicity, let us assume all the transistors have the same amount of drain-source current mismatch shown in Equation 5.14 where the standard deviation of ϵ for every transistor has the same value. The corresponding mismatch terms ϵ are denoted $\epsilon_j, j = 1, \cdots, n$ for the diode-connected transistors of the Y direction input and $\epsilon_{i,j}, i = 1, \cdots, m, j = 1, \cdots, n$ for the transistors in the product core as shown in Figure 5.1.

In Figure 5.2, probabilities are represented by corresponding currents and $\sum_{l=1}^{n} I_{y,l}$

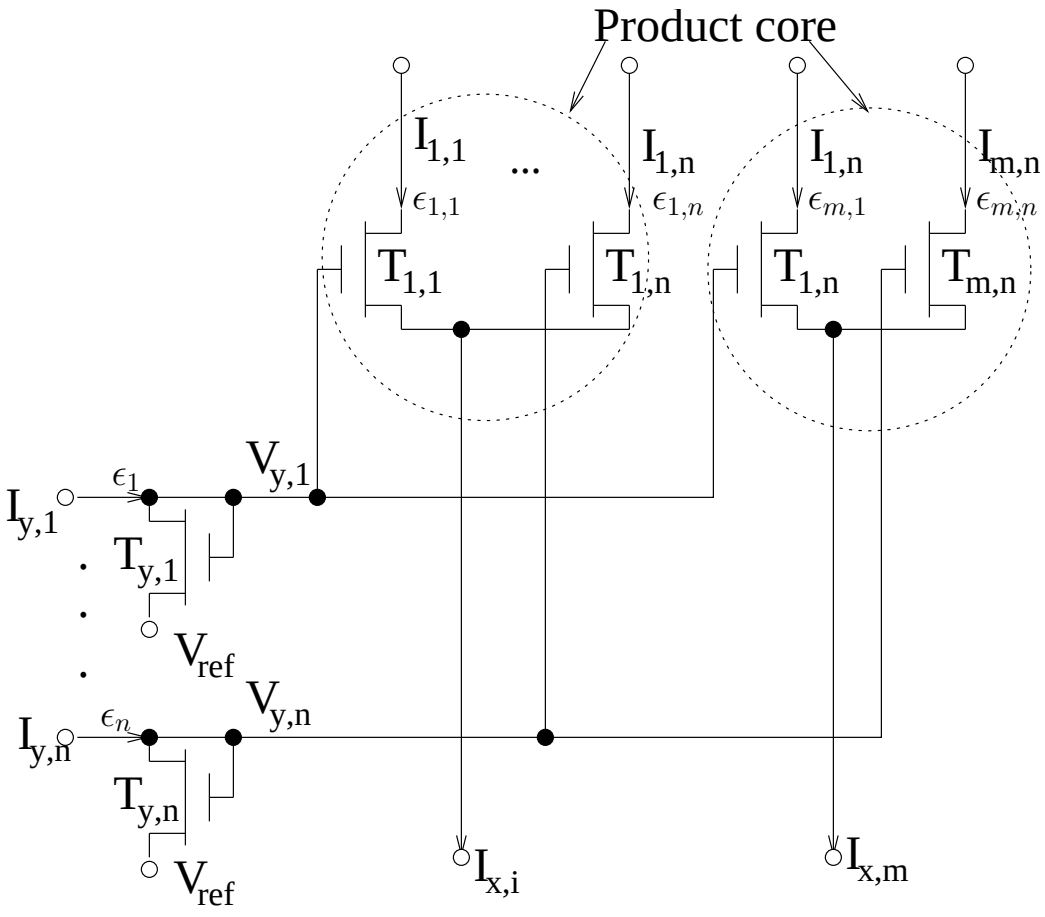

Figure 5.1. Circuit mismatch.

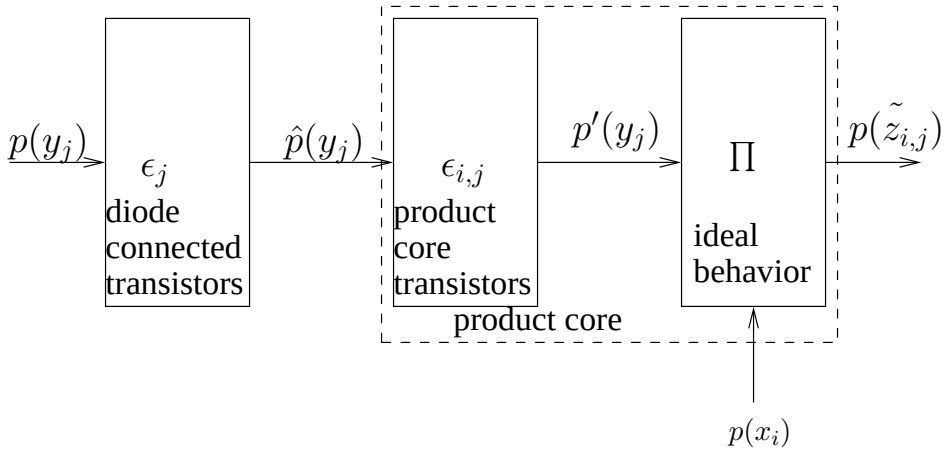

Figure 5.2. Propagation of the mismatch errors.

represent the probability 1 and Equation 5.15 is true. Following Lustenberger's approach, from Figure 5.2, which shows the calculation process of mismatch errors, we can see that due to the mismatch of the diode-connected transistors, the equivalent probability input distribution can be expressed by Equation 5.16. Since Equation 5.17 is true, the mismatch of the transistor $T_{i,j}$ can be viewed as an error of $\frac{U_T \ln(1+\epsilon_{i,j})}{\kappa}$ in its gate voltage $V_{y,j}$. Using Equation 5.17 again, we can propagate the mismatch of transistor $T_{i,j}$ to a factor of $1+\epsilon_{i,j}$ to the input current $I_{y,j}$. As a result, for some fixed i, the total mismatch error can be expressed by modifying the input distribution from $p(y_j)$ to $p'(y_j)$, which is shown in Equation 5.18. Using Equation 5.18, Lustenberger did a high level simulation for the (44,22,8) low-density parity-check code. However, the high level simulation is still too time consuming. Now, we proceed further to reach a much more simple result.

$$\begin{aligned} p(y_j) &= \frac{I_{y,j}}{\sum_{l=1}^{n} I_{y,l}} \\ &= \frac{p_(y_j)}{\sum_{l=1}^{n} p(y_l)} \end{aligned} \tag{5.15}$$

$$\begin{aligned} \hat{p}(y_j) &= \frac{(1+\epsilon_j) I_{y,j}}{\sum_{l=1}^{n} (1+\epsilon_l) I_{y,l}} \\ &= \frac{(1+\epsilon_j) p(y_j)}{\sum_{l=1}^{n} (1+\epsilon_l) p(y_l)} \end{aligned} \tag{5.16}$$

$$\begin{aligned} I_{DS}(1+X) &= I_{0n}e^{\frac{\kappa V_G - V_S}{U_T}}(1+X) \\ &= I_{0n}e^{\frac{\kappa(V_G + \frac{U_T \ln(1+X)}{\kappa}) - V_S}{U_T}} \end{aligned} \tag{5.17}$$

$$\begin{aligned} p'(y_j) &= \frac{(1+\epsilon_{i,j})\hat{p}(y_j)}{\sum_{k=1}^{n}(1+\epsilon_{i,k})\hat{p}(y_k)} \\ &= \frac{(1+\epsilon_{i,j})(1+\epsilon_j)p(y_j)}{\sum_{k=1}^{n}(1+\epsilon_{i,k})(1+\epsilon_k)p(y_k)} \end{aligned} \tag{5.18}$$

From Equation 5.18, we have the ratio of the probability $p'(y_j)$ to $p(y_j)$ as shown in Equation 5.19.

$$\frac{p'(y_j)}{p(y_j)} = \frac{(1+\epsilon_{i,j})(1+\epsilon_j)}{\sum_{k=1}^{n}(1+\epsilon_{i,k})(1+\epsilon_k)p(y_k)} \tag{5.19}$$

From Table 5.1, we know the standard deviation of the drain-source current mismatch, σ, is usually much smaller than 1 ($\sigma \ll 1$). By using Equation 5.9 on Equation 5.19, we obtain Equation 5.20.

$$\begin{aligned} \frac{p'(y_j)}{p(y_j)} &\approx \frac{(1+\sqrt{2}\epsilon_j)}{\sum_{k=1}^{n}(1+\sqrt{2}\epsilon_k)p(y_k)} \\ &= \frac{(1+\sqrt{2}\epsilon_j)}{\sum_{k=1,k\neq j}^{n}(1+\sqrt{2}\epsilon_k)p(y_k) + (1+\sqrt{2}\epsilon_j)p(y_j)} \\ &= \frac{(1+\sqrt{2}\epsilon_j)}{\sum_{k=1,k\neq j}^{n}p(y_k) + \sum_{k=1,k\neq j}^{n}p(y_k)\sqrt{2}\epsilon_k + (1+\sqrt{2}\epsilon_j)p(y_j)} \end{aligned} \tag{5.20}$$

Since the sum of all the input probabilities equals 1, Equation 5.21 exists.

$$\begin{aligned} \sum_{k=1}^{n} p(y_k) &= 1 \\ \sum_{k=1,k\neq j}^{n} p(y_k) &= 1 - p(y_j) \end{aligned} \tag{5.21}$$

By substituting Equation 5.21 into Equation 5.20, Equation 5.22 is derived.

$$\frac{p'(y_j)}{p(y_j)} = \frac{(1+\sqrt{2}\epsilon_j)}{1 - p(y_j) + \sum_{k=1,k\neq j}^{n}p(y_k)\sqrt{2}\epsilon_k + (1+\sqrt{2}\epsilon_j)p(y_j)} \tag{5.22}$$

Using Equation 5.3, Equation 5.23 is derived. As a result, when all the $p(y_k)(k \neq j)$ equal $\frac{1-p(y_j)}{n-1}$, $\sum_{k=1,k\neq j}^{n}p(y_k)\sqrt{2}\epsilon_k$ has the minimum value and when one of the $p(y_k)$

equal $1-p(y_j)$ and all the other $p(y_k)$ equal zero, $\sum_{k=1,k\neq j}^{n} p(y_k)\sqrt{2}\epsilon_k$ has the maximum value just as shown in Equations 5.24 and 5.25. By combining Equations 5.24 and 5.25 Equation 5.26 is derived. The value of m is between $\sqrt{\frac{1}{n-1}}$ and 1 and the value of m does not depend on $p(y_j)$. It only depends on how the other $p(y_k)(k=1,\cdots,n,k\neq j)$ are distributed. Also, it shows that if the cell becomes large, mismatch effects may become small. In the following analysis, we choose the worst case $m=1$ to do the analysis.

$$\sum_{k=1,k\neq j}^{n} p(y_k)\sqrt{2}\epsilon_k = \sqrt{\sum_{k=1,k\neq j}^{n} p^2(y_k)\sqrt{2}\epsilon_l} \tag{5.23}$$

$$\begin{aligned}\sum_{k=1,k\neq j}^{n} p(y_k)\sqrt{2}\epsilon_k &= \sqrt{\sum_{k=1,k\neq j}^{n} (\frac{1-p(y_j)}{n-1})^2\sqrt{2}\epsilon_l} \\ &= (1-p(y_j))\sqrt{\frac{1}{n-1}}\sqrt{2}\epsilon_l \\ &\quad (when\ p(y_k)=\frac{1-p(y_j)}{n-1}\ (k=1,\cdots,n;\ k\neq j))\end{aligned} \tag{5.24}$$

$$\begin{aligned}&\sum_{k=1,k\neq j}^{n} p(y_k)\sqrt{2}\epsilon_k = (1-p(y_j))\sqrt{2}\epsilon_l \\ &(when\ p(y_l)=1-p(y_j)\ p(y_k)=0\ (k=1,\cdots,n;\ k\neq j;\ k\neq l))\end{aligned} \tag{5.25}$$

$$\begin{aligned}&(1-p(y_j))\sqrt{\frac{1}{n-1}}\sqrt{2}\epsilon_l \leq \sum_{k=1,k\neq j}^{n} p(y_k)\sqrt{2}\epsilon_k \leq (1-p(y_j))\sqrt{2}\epsilon_l \\ &\sum_{k=1,k\neq j}^{n} p(y_k)\sqrt{2}\epsilon_k = (1-p(y_j))m\sqrt{2}\epsilon_l\ (\sqrt{\frac{1}{n-1}}\leq m\leq 1)\end{aligned} \tag{5.26}$$

If we choose $m=1$ and substitute Equation 5.26 into Equation 5.22, we have Equation 5.27.

$$\begin{aligned}\frac{p'(y_j)}{p(y_j)} &= \frac{(1+\sqrt{2}\epsilon_j)}{1-p(y_j)+\sqrt{2}\epsilon_l(1-p(y_j))+(1+\sqrt{2}\epsilon_j)p(y_j)} \\ &= \frac{(1+\sqrt{2}\epsilon_j)}{(1+\sqrt{2}\epsilon_l)(1-p(y_j))+(1+\sqrt{2}\epsilon_j)p(y_j)} \\ &= 1+\frac{(1+\sqrt{2}\epsilon_j)(1-p(y_j))-(1+\sqrt{2}\epsilon_l)(1-p(y_j))}{(1+\sqrt{2}\epsilon_l)(1-p(y_j))+(1+\sqrt{2}\epsilon_j)p(y_j)}\end{aligned}$$

$$= 1 + \frac{(\sqrt{2}\epsilon_j - \sqrt{2}\epsilon_l)(1 - p(y_j))}{1 + \sqrt{2}\epsilon_l(1 - p(y_j)) + \sqrt{2}\epsilon_j p(y_j)} \tag{5.27}$$

By using Equation 5.5 in the forward direction on Equation 5.27 and regarding $-(\sqrt{2}\epsilon_l(1 - p(y_j)) + \sqrt{2}\epsilon_j p(y_j))$ as the X in Equation 5.5, Equation 5.28 is derived.

$$\frac{p'(y_j)}{p(y_j)} \approx 1 + (1 - p(y_j))(\sqrt{2}\epsilon_j - \sqrt{2}\epsilon_l)(1 - \sqrt{2}\epsilon_l(1 - p(y_j)) - \sqrt{2}\epsilon_j p(y_j)) \tag{5.28}$$

By using Equation 5.8 in the forward direction on Equation 5.28 and regarding $(1 - p(y_j))(\sqrt{2}\epsilon_j - \sqrt{2}\epsilon_l)$ as the X and regarding $-\sqrt{2}\epsilon_l(1 - p(y_j)) - \sqrt{2}\epsilon_j p(y_j)$ as the Y in Equation 5.8, Equation 5.29 is derived.

$$\frac{p'(y_j)}{p(y_j)} \approx 1 + (1 - p(y_j))(\sqrt{2}\epsilon_j - \sqrt{2}\epsilon_l) \tag{5.29}$$

Finally, by using Equation 5.3 on Equation 5.29, Equation 5.30 is derived.

$$\frac{p'(y_j)}{p(y_j)} = 1 + (1 - p(y_j))2\epsilon_j \tag{5.30}$$

From Equation 5.30, we can see that the larger $p(y_j)$ is, the less it is affected by mismatch. For the X direction input, similar analysis generates a similar result shown in Equation 5.31.

$$\frac{p'(x_j)}{p(x_j)} = 1 + (1 - p(x_j))2\epsilon_j \tag{5.31}$$

From Figure 4.23, we can see that usually the channel information comes from the X input. Now, let us analyze the X direction first. From Chapter 2, we know that the probability of sending a 1 and receiving y is the channel information that is expressed by Equation 2.25 (shown again for convenience as Equation 5.32.) in which $y = x + \epsilon_e$ ($x = \pm 1$), ϵ_e is the channel noise and its standard deviation is σ_e. Using the probability of sending a 0 and receiving y to do analysis is similar.

$$p(x_j) = \frac{1}{1 + e^{\frac{-4y}{N_0}}} \tag{5.32}$$

By substituting Equation 5.32 into Equation 5.31, the equivalent channel information after considering mismatch effect can be expressed by Equation 5.33.

$$\begin{aligned} p'(x_j) &= p(x_j)(1 + (1 - p(x_j))2\epsilon_j) \\ &= \frac{1}{1 + e^{\frac{-4y}{N_0}}}\left(1 + \left(1 - \frac{1}{1 + e^{\frac{-4y}{N_0}}}\right)2\epsilon_j\right) \end{aligned}$$

$$
\begin{aligned}
&= \frac{1}{1+e^{\frac{-4y}{N_0}}}\left(1+\frac{e^{\frac{-4y}{N_0}}}{1+e^{\frac{-4y}{N_0}}}2\epsilon_j\right)\\
&= \frac{1}{1+e^{\frac{-4y}{N_0}}}\left(\frac{1+e^{\frac{-4y}{N_0}}(1+2\epsilon_j)}{1+e^{\frac{-4y}{N_0}}}\right)\\
&= \frac{1+e^{\frac{-4y}{N_0}}+e^{\frac{-4y}{N_0}}2\epsilon_j}{(1+e^{\frac{-4y}{N_0}})^2}
\end{aligned}
\tag{5.33}
$$

Because $1+e^{\frac{-4y}{N_0}} > e^{\frac{-4y}{N_0}}$, we can use Equation 5.6 in the backward direction on Equation 5.33 by regarding $1+e^{\frac{-4y}{N_0}}$ as a and $e^{\frac{-4y}{N_0}}$ as b in Equation 5.6. As a result, Equation 5.34 can be derived.

$$
\begin{aligned}
p'(x_j) &\approx \frac{1}{1+e^{\frac{-4y}{N_0}}-e^{\frac{-4y}{N_0}}2\epsilon_j}\\
&= \frac{1}{1+e^{\frac{-4y}{N_0}}(1-2\epsilon_j)}
\end{aligned}
\tag{5.34}
$$

Using Equation 5.12 in backward direction on Equation 5.34 by regarding $-2\epsilon_j$ as the X in Equation 5.12, Equation 5.35 is derived.

$$
\begin{aligned}
p'(x_j) &\approx \frac{1}{1+e^{\frac{-4y}{N_0}}e^{-2\epsilon_j}}\\
&= \frac{1}{1+e^{\frac{-4(y+\frac{N_0}{2}\epsilon_j)}{N_0}}}\\
&= \frac{1}{1+e^{\frac{-4(\pm 1+\epsilon_e+\sigma_e^2\epsilon_j)}{N_0}}} \quad (\sigma_e^2=\frac{N_0}{2} \text{ from Equation 2.23})
\end{aligned}
\tag{5.35}
$$

From Equation 5.35, we can see that the mismatch effect can be considered as a kind of noise. Notice that for a certain noise, σ_e and N_0 are constant. As a result, we can see that the standard deviation of the new noise is $\sqrt{\sigma_e^2+\sigma_e^4\sigma_j^2}$ where σ_j is the standard deviation of the drain-source current mismatch for one transistor. Now, we see that the performance loss in dB due to mismatch for X input can be expressed by Equation 5.36.

$$
10\log\frac{\sigma_e^2+\sigma_e^4\sigma_j^2}{\sigma_e^2} = 10\log\left(1+\sigma_e^2\sigma_j^2\right) \tag{5.36}
$$

Actually, we can observe the mismatch from the voltage (log-domain) perspective and get the same result. When we choose $m=1$ and use Equation 5.27, the situation is just

the same as having only two inputs, as shown in Figure 5.3. By using Equation 2.25 and Equation 4.13, we have the voltage difference shown in Equation 5.37 without considering mismatch.

$$\begin{aligned} \delta V &= \frac{U_T}{\kappa} \ln \frac{p_1}{p_0} \\ &= \frac{U_T}{\kappa} \frac{4y}{N_0} \end{aligned} \tag{5.37}$$

If mismatch is considered, the voltage difference is then expressed by Equation 5.38. By using Equation 5.5, Equation 5.9, and Equation 5.13 on Equation 5.38, Equation 5.39 can be derived. From Equation 5.39, the same result shown in Equation 5.36 can be derived.

$$\begin{aligned} \delta V &= \frac{U_T}{\kappa} \ln \frac{p_1'}{p_0'} \\ &= \frac{U_T}{\kappa} \ln \frac{p_1(1+\sqrt{2}\epsilon_1)}{p_0(1+\sqrt{2}\epsilon_0)} \\ &= \frac{U_T}{\kappa} (\frac{4y}{N_0} + \ln \frac{1+\sqrt{2}\epsilon_1}{1+\sqrt{2}\epsilon_0}) \end{aligned} \tag{5.38}$$

$$\begin{aligned} \delta V &= \frac{U_T}{\kappa} (\frac{4y}{N_0} + \ln \frac{1+\sqrt{2}\epsilon_1}{1+\sqrt{2}\epsilon_0}) \\ &\approx \frac{U_T}{\kappa} (\frac{4y}{N_0} + \ln (1+\sqrt{2}\epsilon_1)(1-\sqrt{2}\epsilon_0)) \text{ (Using Equation 5.5)} \\ &\approx \frac{U_T}{\kappa} (\frac{4y}{N_0} + \ln (1+2\epsilon_1)) \text{ (Using Equation 5.9)} \\ &\approx \frac{U_T}{\kappa} (\frac{4y}{N_0} + 2\epsilon_1) \text{ (Using Equation 5.13)} \end{aligned}$$

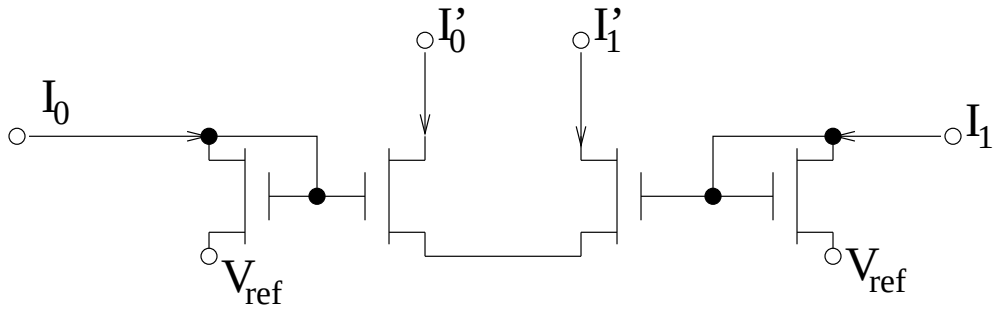

Figure 5.3. Mismatch of two pair of transistors.

$$= \frac{U_T}{\kappa}\frac{4(\pm 1+\epsilon_e+\sigma_e^2\epsilon_1)}{N_0}\ (N_0=2\sigma_e^2) \tag{5.39}$$

Now, we discuss the total performance loss due to transistor mismatch. From Equation 5.35 and Equation 5.39, we know that the mismatch effect is not large. As a result, when the decision is quite clear if mismatch is not considered, then the decision cannot be changed even if mismatch is considered. Only when the decision is not quite clear is there a concern about the mismatch effect. From Figure 4.23, we know that the channel information is typically provided from the X direction where the context information is provided from the Y direction. When the decision is not clear, the context information is nearly uniformly distributed, especially when SNR is high. As a result, the context information provided to the next trellis section is nearly equal to the channel information provided by the current stage. By propagating the mismatch effect that is generated by the context information to the next trellis section back to the current X direction channel information, the mismatch effect is doubled. Also, notice that each building block uses not only a product cell, but also a normalization cell. The analysis of the normalization cell is similar to the analysis of the product cell and the normalization cell is also affected by mismatch. Considering all these mismatch effects, the final mismatch effect is shown in Equation 5.40 (Notice that the equivalent noise is $\sqrt{3}\sigma_e^2\epsilon$ because of the Gaussian operation).

$$10\log\frac{\sigma_e^2+3\sigma_e^4\sigma_j^2}{\sigma_e^2} = 10\log\left(1+3\sigma_e^2\sigma_j^2\right) \tag{5.40}$$

A better way of analyzing the total mismatch effect is observing it from the log-domain (voltage operation). What two kind of decoders are widely used? One uses block-wise ML decision rule. Another uses bit-wise MAP decision rule. For high SNR, the two kinds of decoders generate nearly the same result. From the previous analysis, we know that for each trellis stage, from the log-domain view, the received information is $\frac{4y_i}{N_0}$ (see Equation 5.37) in which y_i is the received value for the ith symbol and $y_i = x_i + \epsilon_e = \pm 1 + \epsilon_e$. (We ignore the constant $\frac{U_T}{\kappa}$ for simplicity.) Also, the mismatch effect for each trellis section is the same. For both the X and Y direction, the mismatch effect is 2ϵ. For the log-domain, the mismatch from the X and Y direction should be added together. Also, the mismatch effect caused by the normalization cell is also 2ϵ and this mismatch effect should also be added. As a result, each trellis section has a mismatch effect of $2\sqrt{3}\epsilon$. Thus, the equivalent information provided by each trellis

section is $\frac{4y_i}{N_0} + 2\sqrt{3}\epsilon = \frac{4(\pm 1+\epsilon_e+\sqrt{3}\sigma_e^2\epsilon)}{N_0}$. As a result, we can again get the performance loss due to mismatch shown in Equation 5.40.

For iterative decoding such as the low-density parity-check code shown in Figure 5.4, we know that the channel information needs to pass through both the variable node (equal gate) and the function node before it is used. The variable node is implemented by an equal gate and the function node is implemented by an XOR gate. Each gate gives a $2\sqrt{3}\epsilon$ mismatch to the channel information. Adding the mismatch together, the total equivalent mismatch effect to the channel information is $2\sqrt{6}\epsilon$. As a result, for low-density parity-check code, the performance loss due to mismatch is shown in Equation 5.41. Actually, for a iterative decoders, the information $\frac{4y_i}{N_0}$ (see Equation 5.37) needs to pass through two building blocks before it is used. As a result, the nonlinear effect needs to be doubled for iterative decoders.

$$10\log\frac{\sigma_e^2 + 6\sigma_e^4\sigma_j^2}{\sigma_e^2} = 10\log\left(1 + 6\sigma_e^2\sigma_j^2\right) \tag{5.41}$$

Notice that Equation 5.40 and Equation 5.41 only show the average performance loss due to mismatch. Because mismatch is fixed for every chip, the mismatch effect for every

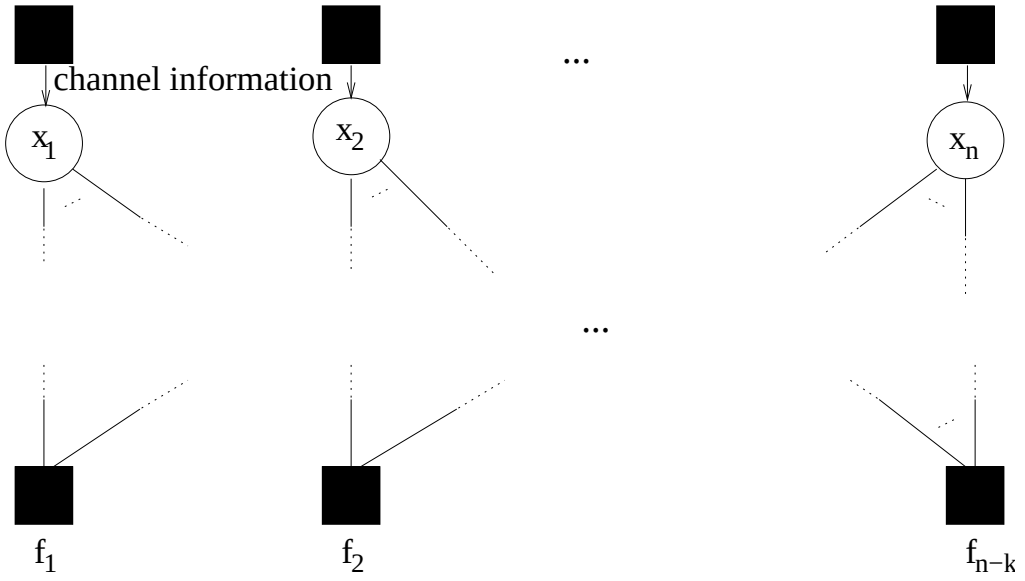

Figure 5.4. The structure of a low-density parity-check code.

chip can be different. Simulation has shown that for some chips, mismatch effect can even improve the performance.

Also, from Equation 5.40 and Equation 5.41, we know the performance loss due to the mismatch effect is not large. Suppose the variance of the channel noise is $\sigma_e^2 = 0.2$, and the standard deviation of the drain-source current mismatch of one transistor, $\sigma_j = 0.1$, then the performance loss using Equation 5.40 is 0.025dB and the performance loss using Equation 5.41 is 0.052dB. Also, when SNR increases, σ_e^2 decreases. The performance loss in dB due to mismatch also decreases. As a result, the average bit-error rate curve after including the mismatch effect is a curve nearly parallel to the ideal bit-error rate curve. Figure 5.5 shows the ideal bit-error rate curve, the simulation result of bit-error rate curve including mismatch effect by using Equation 5.18 and the fitting curve by using Equation 5.41 of the Hamming (8,4) decoder. They match very well, especially at high SNR.

5.3 Internal Noise

There are two kind of noise for MOS transistors, thermal noise and flicker noise. In the following, we discuss them separately.

5.3.1 Thermal Noise

The variance of the thermal noise of MOS transistors is usually expressed by the drain-source current in the frequency domain by Equation 5.42 [29] [56].

$$I_n^2(f) = 4kT(\frac{2}{3})g_m \tag{5.42}$$

For MOS transistors working in the weak inversion region, $g_m = \frac{\kappa}{U_T} I_{ds}$. Assuming that the noise bandwidth is f_x, then Equation 5.43 is true in which q is the electrical quantity of one electron and $q = 1.603 * 10^{-19} C$.

$$\begin{aligned} I_n^2 &= 4kT(\frac{2}{3})g_m f_x \\ &= 4kT(\frac{2}{3}) f_x \frac{\kappa}{U_T} I_{ds} \\ &= \frac{8}{3}\kappa f_x q I_{ds} \ (U_T = \frac{kT}{q}) \\ I_n &= \sqrt{\frac{8}{3}\kappa f_x q}\sqrt{I_{ds}} \end{aligned} \tag{5.43}$$

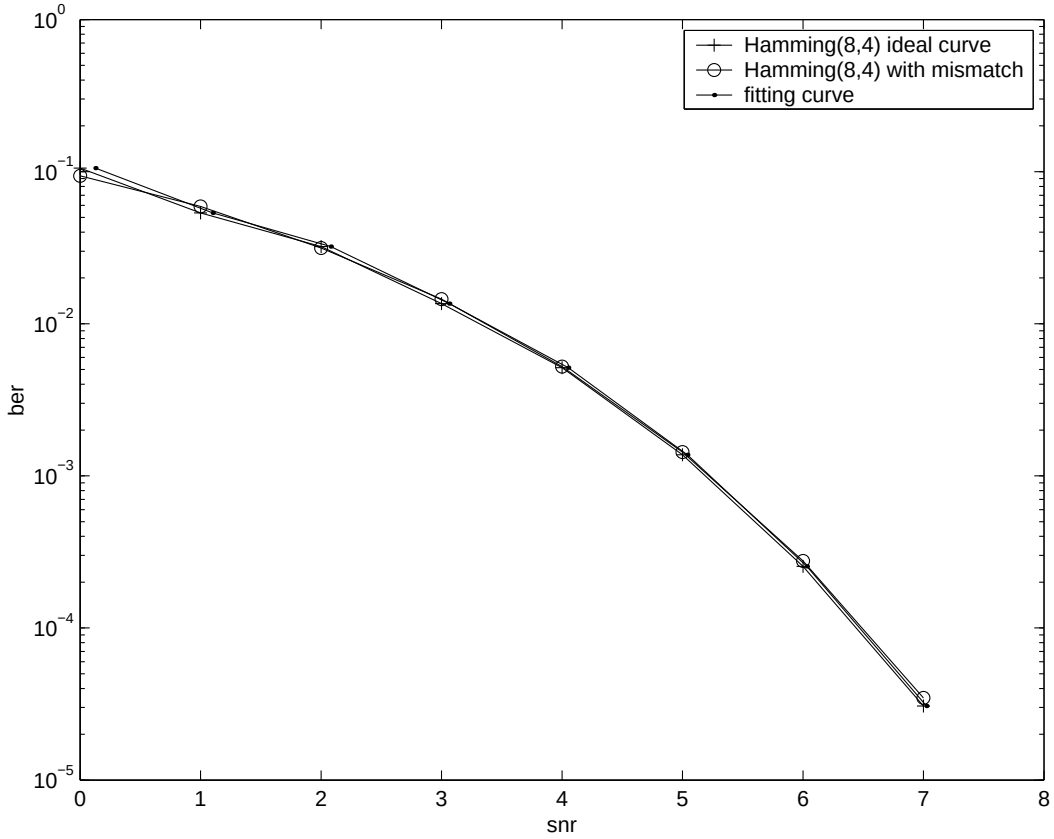

Figure 5.5. The mismatch effect of the tail-biting extended Hamming (8,4) decoder.

Now, the drain-source current for one transistor after considering thermal noise can be expressed by Equation 5.44 in which the standard deviation of ϵ is $\sigma = \sqrt{\frac{8}{3}\kappa f_x q}$.

$$I'_{ds} = I_{ds} + \sqrt{I_{ds}}\,\epsilon \tag{5.44}$$

Now, following the approach of the mismatch analysis, Equation 5.45 can be derived for the X direction in which I_i is the drain-source current of transistor T_i before mismatch is considered and I'_i is the drain-source current of transistor T_i after mismatch is considered.

$$\begin{aligned} I'_i &= I_i + \sqrt{I_i}\sqrt{2}\epsilon_i \\ &= I_i + \sqrt{2I_i}\epsilon_i (i = 1, ..., m) \end{aligned} \tag{5.45}$$

As a result, the probability distribution after considering thermal noise can be expressed by Equation 5.46.

$$\begin{aligned} p'_j &= \frac{I_j + \sqrt{2I_j}\epsilon_j}{\sum_{i=1}^{m} I_i + \sqrt{2I_i}\epsilon_i} \\ &= \frac{I_j + \sqrt{2I_j}\epsilon_j}{I_u + \sqrt{2I_j}\epsilon_j + \sum_{i=1,i\neq j}^{m}\sqrt{2I_i}\epsilon_i} \\ &= \frac{p_j + \frac{\sqrt{2I_j}}{I_u}\epsilon_j}{1 + \frac{\sqrt{2I_j}}{I_u}\epsilon_j + \frac{1}{I_u}\sum_{i=1,i\neq j}^{m}\sqrt{2I_i}\epsilon_i} \left(p_j = \frac{I_j}{I_u}\, and \sum_{i=1}^{m} I_i = I_u\right) \end{aligned} \tag{5.46}$$

By using Equation 5.5 on Equation 5.46 and regarding $-(\frac{\sqrt{2I_j}}{I_u}\epsilon_j + \frac{1}{I_u}\sum_{i=1,i\neq j}^{m}\sqrt{2I_i}\epsilon_i)$ as the X in Equation 5.5, Equation 5.47 can be derived.

$$\begin{aligned} p'_j &\approx (p_j + \frac{\sqrt{2I_j}}{I_u}\epsilon_j)(1 - \frac{\sqrt{2I_j}}{I_u}\epsilon_j - \frac{1}{I_u}\sum_{i=1,i\neq j}^{m}\sqrt{2I_i}\epsilon_i) \\ &= p_j(1 - \frac{\sqrt{2I_j}}{I_u}\epsilon_j - \frac{1}{I_u}\sum_{i=1,i\neq j}^{m}\sqrt{2I_i}\epsilon_i) \\ &\quad + \frac{\sqrt{2I_j}}{I_u}\epsilon_j(1 - \frac{\sqrt{2I_j}}{I_u}\epsilon_j - \frac{1}{I_u}\sum_{i=1,i\neq j}^{m}\sqrt{2I_i}\epsilon_i) \end{aligned} \tag{5.47}$$

By using Equation 5.8 on Equation 5.47 and regarding $\frac{\sqrt{2I_j}}{I_u}$ as the X, $-\frac{\sqrt{2I_j}}{I_u}\epsilon_j - \frac{1}{I_u}\sum_{i=1,i\neq j}^{m}\sqrt{2I_i}\epsilon_i$ as the Y in Equation 5.8, Equation 5.48 can be derived.

$$p'_j \approx p_j(1 - \frac{\sqrt{2I_j}}{I_u}\epsilon_j - \frac{1}{I_u}\sum_{i=1,i\neq j}^{m}\sqrt{2I_i}\epsilon_i) + \frac{\sqrt{2I_j}}{I_u}\epsilon_j$$

$$
\begin{aligned}
&= p_j(1-\frac{1}{I_u}\sum_{i=1,i\neq j}^{m}\sqrt{2I_i}\epsilon_i)+\frac{\sqrt{2I_j}\epsilon_j}{I_u}(1-p_j)\\
&= p_j(1-\frac{\sqrt{2}}{\sqrt{I_u}}\sum_{i=1,i\neq j}^{m}\sqrt{p_i}\epsilon_i)+\frac{\sqrt{2}\epsilon_j}{\sqrt{I_u}}\sqrt{p_j}(1-p_j) \qquad (5.48)
\end{aligned}
$$

Using Equation 5.3 on Equation 5.48, Equation 5.49 can be derived.

$$
\begin{aligned}
p'_j &= p_j(1-\frac{\sqrt{2}\epsilon_k}{\sqrt{I_u}}\sqrt{\sum_{i=1,i\neq j}^{m}(\sqrt{p_i})^2})+\frac{\sqrt{2}\epsilon_j}{\sqrt{I_u}}\sqrt{p_j}(1-p_j)\\
&= p_j(1-\frac{\sqrt{2}\epsilon_k}{\sqrt{I_u}}\sqrt{\sum_{i=1,i\neq j}^{m}p_i})+\frac{\sqrt{2}\epsilon_j}{\sqrt{I_u}}\sqrt{p_j}(1-p_j)\\
&= p_j(1-\frac{\sqrt{2}\epsilon_k}{\sqrt{I_u}}\sqrt{1-p_j})+\frac{\sqrt{2}\epsilon_j}{\sqrt{I_u}}\sqrt{p_j}(1-p_j)\\
&= p_j(1-k_1\sqrt{1-p_j})+k_2\sqrt{p_j}(1-p_j)\ (k_1=\frac{\sqrt{2}\epsilon_k}{\sqrt{I_u}};\ k_2=\frac{\sqrt{2}\epsilon_j}{\sqrt{I_u}}) \qquad (5.49)
\end{aligned}
$$

By substituting Equation 5.32 into Equation 5.49, Equation 5.50 can be derived.

$$
\begin{aligned}
p'(j) &= p_j(1-k_1\sqrt{1-p_j})+k_2\sqrt{p_j}(1-p_j)\\
&= \frac{1}{1+e^{\frac{-4y}{N_0}}}\left(1-k_1\sqrt{1-\frac{1}{1+e^{\frac{-4y}{N_0}}}}\right)+k_2\frac{1}{\sqrt{1+e^{\frac{-4y}{N_0}}}}(1-\frac{1}{1+e^{\frac{-4y}{N_0}}})\\
&= \frac{1}{1+e^{\frac{-4y}{N_0}}}(1-\frac{k_1e^{\frac{-2y}{N_0}}}{\sqrt{1+e^{\frac{-4y}{N_0}}}})+k_2\frac{1}{\sqrt{1+e^{\frac{-4y}{N_0}}}}\frac{e^{\frac{-4y}{N_0}}}{1+e^{\frac{-4y}{N_0}}}\\
&= \frac{1}{1+e^{\frac{-4y}{N_0}}}(1-\frac{k_1e^{\frac{-2y}{N_0}}}{\sqrt{1+e^{\frac{-4y}{N_0}}}}+\frac{k_2e^{\frac{-4y}{N_0}}}{\sqrt{1+e^{\frac{-4y}{N_0}}}}) \qquad (5.50)
\end{aligned}
$$

By using Equation 5.5 in the backward direction on Equation 5.50 and regarding $-\frac{k_1e^{\frac{-2y}{N_0}}}{\sqrt{1+e^{\frac{-4y}{N_0}}}}+\frac{k_2e^{\frac{-4y}{N_0}}}{\sqrt{1+e^{\frac{-4y}{N_0}}}}$ as the X in Equation 5.5, Equation 5.51 can be derived.

$$
\begin{aligned}
p'(j) &\approx \frac{1}{1+e^{\frac{-4y}{N_0}}}\frac{1}{1+\frac{k_1e^{\frac{-2y}{N_0}}}{\sqrt{1+e^{\frac{-4y}{N_0}}}}-\frac{k_2e^{\frac{-4y}{N_0}}}{\sqrt{1+e^{\frac{-4y}{N_0}}}}}\\
&= \frac{1}{1+e^{\frac{-4y}{N_0}}+k_1e^{\frac{-2y}{N_0}}\sqrt{1+e^{\frac{-4y}{N_0}}}-k_2e^{\frac{-4y}{N_0}}\sqrt{1+e^{\frac{-4y}{N_0}}}} \qquad (5.51)
\end{aligned}
$$

In Equation 5.51, $k_1=\frac{\sqrt{2}\epsilon_k}{I_u}$, $k_2=\frac{\sqrt{2}\epsilon_j}{I_u}$ and the standard deviation of ϵ_k and ϵ_j are both $\sqrt{\frac{8}{3}\kappa f_x q}$. Thus, the standard deviation of k_1 and k_2 are both $\frac{\sqrt{2}\sqrt{\frac{8}{3}\kappa f_x q}}{\sqrt{I_u}}$. Usually, this

standard deviation is very small. For example, for $\kappa = 0.6$, noise bandwidth $f_x = 10MHZ$ and $I_u = 100nA$, the standard deviation is only $6.02 * 10^{-3}$. As a result, the performance loss due to thermal noise is negligible compared with the mismatch effect.

5.3.2 Flicker Noise

The variance of flicker noise of MOS transistors is usually expressed by the gate-source voltage in the frequency domain shown in Equation 5.52 [29] [56] in which K is a process dependent constant and usually has a magnitude around $10^{-25}V^2F$.

$$V_n^2(f) = \frac{K}{WLC_{ox}f} \tag{5.52}$$

Assuming the noise bandwidth is f_x and the flicker noise exists between 1 and f_x, the total flicker noise can be expressed by Equation 5.53.

$$\begin{aligned} V_n^2 &= \int_0^{f_x} \frac{K}{WLC_{ox}f}\,\mathrm{d}f \\ V_n^2 &= \frac{K}{WLC_{ox}} \ln f_x \\ V_n &= \sqrt{\frac{K}{WLC_{ox}} \ln f_x} \end{aligned} \tag{5.53}$$

As a result, the corresponding drain-source current flicker noise can be expressed by Equation 5.54.

$$\begin{aligned} I_n &= g_m\sqrt{\frac{K}{WLC_{ox}} \ln f_x} \\ &= \frac{\kappa}{U_T}\sqrt{\frac{K}{WLC_{ox}} \ln f_x} I_{ds} \ (g_m = \frac{\kappa}{U_T} I_{ds}) \end{aligned} \tag{5.54}$$

The total drain-source current after considering flicker noise can be expressed as the following equation in which the standard deviation of ϵ equals $\frac{\kappa}{U_T}\sqrt{\frac{K}{WLC_{ox}} \ln f_x}$ that is independent of I_{ds}.

$$I'_{ds} = (1 + \epsilon)I_{ds} \tag{5.55}$$

Following the approach of mismatch, it is easy to know that the flicker noise effect is equal to an additional noise of $3\sigma_e^2\epsilon$ to $6\sigma_e^2\epsilon$ to the channel where σ_e is the standard deviation of the channel noise. For typical applications, the standard deviation of ϵ is quite small. For example, if $\kappa = 0.6$, $U_T = 0.0258V$, $K = 10^{-24}V^2F$, $W = L = 2um$, $C_{ox} = 1.9 * 10^{-3}pF/(um)^2$, and $f_x = 10MHz$, the standard deviation of ϵ is only $1.07 * 10^{-3}$. This value is also quite small compared with the mismatch effect. As a result, the performance due to flicker noise is also negligible.

5.4 Channel Length Modulation

Another effect that needs to be discussed is channel length modulation. From Figure 4.9, we can see that for circuits working under weak inversion, the drain-source voltage of the transistors in the product core are nearly the same. However, Table 5.2 shows the Early voltage of MOS transistors using MOSIS $0.5um$ technology and using width and length $3um$ [73]. It shows that the Early voltage is not constant. The larger the drain-source current, the larger the early voltage. For product core transistors working in strong inversion or the moderate region, from Figure 4.9, we can see that the larger the drain-source current, the smaller the drain-source voltage because the PMOS transistor that provides the current needs more gate-source voltage to provide the current. As a result, no matter which region the transistor works, the larger the drain-source current, the smaller it is scaled up by channel length modulation. Assuming that the channel length modulation coefficient λ is proportional to I_{ds}^{α} where α is a negative number larger than -1 and the drain-source current after considering the channel length modulation effect is $I'_{ds} = I_{ds}(1+\lambda)$, the channel length modulation coefficient can be expressed by Equation 5.56 in which I_u is the reference current that represents probability 1 and k_u is a small constant around 0.01 making $\lambda = 0.01$ when $I_{ds} = I_u$.

$$\lambda = k_1 I_{ds}^{\alpha} \ (k_1 = k_u I_u^{-\alpha}) \tag{5.56}$$

From Figure 4.9, we know the channel length modulation effect is much smaller for the current mirrors in the bottom of the figure compared with the product core circuit. When only the channel length modulation effect for the product core is considered, the probability distribution can be expressed by Equation 5.57.

Table 5.2. Early effect.

NMOS		*PMOS*	
Drain-source current	Early voltage	Drain-source current	Early voltage
62nA	86V	23.7nA	154V
2.4uA	123V	1.15uA	157V
		4.89uA	181V
11.2uA	153V	11.2uA	183V

$$
\begin{aligned}
p'_j &= \frac{I_j(1+k_1 I_j^{\alpha})}{\sum_{i=1}^{n} I_i(1+k_1 I_i^{\alpha})} \\
&= \frac{I_j + k_1 I_j^{\alpha+1}}{I_u + k_1 I_j^{\alpha+1} + \sum_{i=1,i\neq j}^{n} k_1 I_i^{\alpha+1}} \\
&= \frac{p_j + k_1 I_u^{\alpha} p_j^{\alpha+1}}{1 + k_1 I_u^{\alpha} \sum_{i=1,i\neq j}^{n} p_i^{\alpha+1} + k_1 I_u^{\alpha} p_j^{\alpha+1}} \; (p_j = \frac{I_j}{I_u} \; and \; \sum_{i=1}^{n} I_i = I_u) \\
&= \frac{p_j + k_u p_j^{\alpha+1}}{1 + k_u \sum_{i=1,i\neq j}^{n} p_i^{\alpha+1} + k_u p_j^{\alpha+1}} \; (k_u = k_1 I_u^{\alpha}) \qquad (5.57)
\end{aligned}
$$

By using Equation 5.5 on Equation 5.57 and regarding $-(k_u \sum_{i=1,i\neq j}^{n} p_i^{\alpha+1} + k_u p_j^{\alpha+1})$ as the X in Equation 5.5, Equation 5.58 can be derived.

$$
\begin{aligned}
p'_j &\approx (p_j + k_u p_j^{\alpha+1})(1 - k_u \sum_{i=1,i\neq j}^{n} p_i^{\alpha+1} - k_u p_j^{\alpha+1}) \\
&= p_j(1 - k_u \sum_{i=1,i\neq j}^{n} p_i^{\alpha+1} - k_u p_j^{\alpha+1}) \\
&\quad + k_u p_j^{\alpha+1}(1 - k_u \sum_{i=1,i\neq j}^{n} p_i^{\alpha+1} - k_u p_j^{\alpha+1}) \qquad (5.58)
\end{aligned}
$$

By using Equation 5.8 on Equation 5.58 and regarding $k_u p_j^{\alpha+1}$ as the X, $-k_u \sum_{i=1,i\neq j}^{n} p_i^{\alpha+1} - k_u p_j^{\alpha+1}$ as the Y in Equation 5.8, Equation 5.59 can be derived.

$$
\begin{aligned}
p'_j &\approx p_j(1 - k_u \sum_{i=1,i\neq j}^{n} p_i^{\alpha+1} - k_u p_j^{\alpha+1}) + k_u p_j^{\alpha+1} \\
&= p_j(1 - k_u \sum_{i=1,i\neq j}^{n} p_i^{\alpha+1}) + k_u p_j^{\alpha+1}(1 - p_j) \qquad (5.59)
\end{aligned}
$$

Because α is a negative number larger than -1, $\alpha + 1 > 0$ and Equation 5.21 is true. When all the $p_i (i = 1, i \neq j)$ all equal $\frac{1-p_j}{n-1}$, $\sum_{i=1,i\neq j}^{n} p_i$ has the maximum value and when one of the p_i equals $1 - p_j$ and all the other p_i equal zero, $\sum_{k=1,k\neq j}^{n} p_i$ has the minimum value. This is shown in Equation 5.60. Thus, using Equation 5.60 on Equation 5.59, Equation 5.59 can be simplified to Equation 5.61

$$(1-p_j)^{\alpha+1} \leq \sum_{i=1,i\neq j}^{n} p_i^{\alpha+1} \leq (n-1)^{-\alpha}(1-p_j)^{\alpha+1} \tag{5.60}$$

$$\begin{aligned} p'_j &= p_j(1-k_2(1-p_j)^{\alpha+1}) + k_u p_j^{\alpha+1}(1-p_j) \\ &\quad (k_2 = mk_u \; and \; 1 \leq m \leq (n-1)^{-\alpha}) \end{aligned} \tag{5.61}$$

By substituting Equation 5.32 into Equation 5.61, Equation 5.62 can be derived.

$$\begin{aligned} p'_j &= \frac{1}{1+e^{\frac{-4y}{N_0}}}(1-k_2(1-\frac{1}{1+e^{\frac{-4y}{N_0}}})^{\alpha+1}) + \frac{k_u}{(1+e^{\frac{-4y}{N_0}})^{\alpha+1}}(1-\frac{1}{1+e^{\frac{-4y}{N_0}}}) \\ &= \frac{1}{1+e^{\frac{-4y}{N_0}}}(1-k_2\frac{e^{\frac{-4(\alpha+1)y}{N_0}}}{(1+e^{\frac{-4y}{N_0}})^{\alpha+1}}) + \frac{k_u}{(1+e^{\frac{-4y}{N_0}})^{\alpha+1}}\frac{e^{\frac{-4y}{N_0}}}{1+e^{\frac{-4y}{N_0}}} \\ &= \frac{1}{1+e^{\frac{-4y}{N_0}}}(1-k_2\frac{e^{\frac{-4(\alpha+1)y}{N_0}}}{(1+e^{\frac{-4y}{N_0}})^{\alpha+1}} + k_u\frac{e^{\frac{-4y}{N_0}}}{(1+e^{\frac{-4y}{N_0}})^{\alpha+1}}) \end{aligned} \tag{5.62}$$

By using Equation 5.5 in the backward direction on Equation 5.62 and regarding $-k_2\frac{e^{\frac{-4(\alpha+1)y}{N_0}}}{(1+e^{\frac{-4y}{N_0}})^{\alpha+1}} + k_u\frac{e^{\frac{-4y}{N_0}}}{(1+e^{\frac{-4y}{N_0}})^{\alpha+1}}$ as the X in Equation 5.5, Equation 5.63 can be derived.

$$\begin{aligned} p'_j &\approx \frac{1}{1+e^{\frac{-4y}{N_0}}}\frac{1}{1+k_2\frac{e^{\frac{-4(\alpha+1)y}{N_0}}}{(1+e^{\frac{-4y}{N_0}})^{\alpha+1}} - k_u\frac{e^{\frac{-4y}{N_0}}}{(1+e^{\frac{-4y}{N_0}})^{\alpha+1}}} \\ &= \frac{1}{1+e^{\frac{-4y}{N_0}} + k_2 e^{\frac{-4(\alpha+1)y}{N_0}}(1+e^{\frac{-4y}{N_0}})^{-\alpha} - k_u e^{\frac{-4y}{N_0}}(1+e^{\frac{-4y}{N_0}})^{-\alpha}} \end{aligned} \tag{5.63}$$

From Equation 5.59, we know k_2 is a constant and it may increase as m increases. As a result, the performance loss due to channel length modulation may increase for large decoders. Usually k_2 and k_u have very small values such as 0.01. The performance loss due to channel length modulation is again small compared with the performance loss due to mismatch.

5.5 Strong Inversion

Could the "sum-product algorithm" be implemented by transistors not working under weak inversion? If the transistors of Figure 5.6 are working under strong inversion, then the following equations exist.

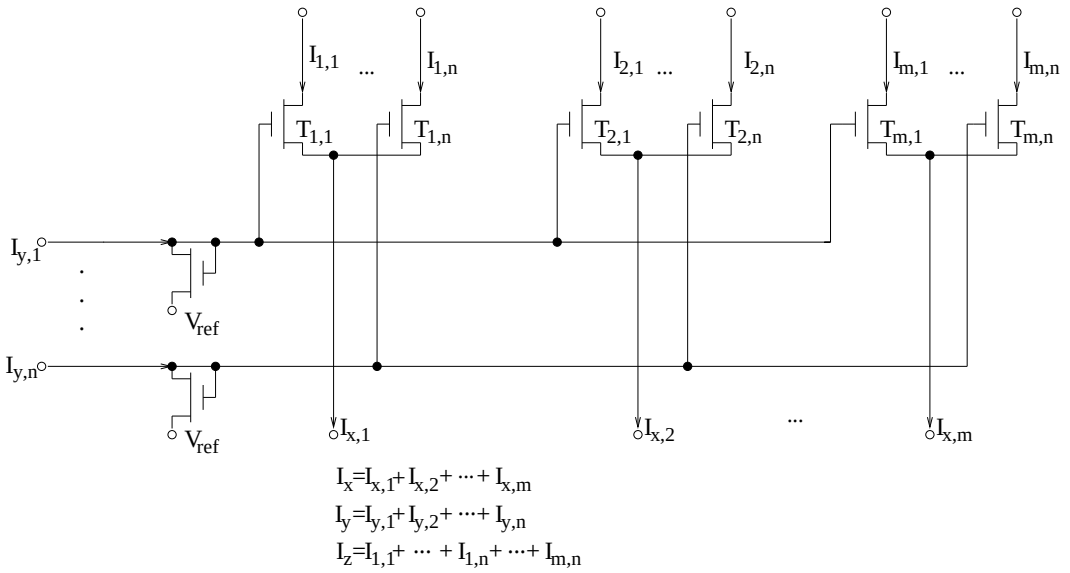

Figure 5.6. Fundamental circuit.

$$I_{i,j}/I_{x,i} = (V_{y,j} - V_{x,i} - V_{th_1})^2 / \sum_{l=1}^{n} (V_{y,l} - V_{x,i} - V_{th_1})^2 \tag{5.64}$$

$$I_{y,j}/I_y = (V_{y,j} - V_{ref} - V_{th_2})^2 / \sum_{l=1}^{n} (V_{y,l} - V_{ref} - V_{th_2})^2 \tag{5.65}$$

Let us define $V_1 = V_{x,i} + V_{th_1}$ and $V_2 = V_{ref} + V_{th_2}$. Then Equation 5.64 and Equation 5.65 are changed to Equation 5.66 and 5.67.

$$I_{i,j}/I_{x,i} = (V_{y,j} - V_1)^2 / \sum_{l=1}^{n} (V_{y,l} - V_1)^2 \tag{5.66}$$

$$I_{y,j}/I_y = (V_{y,j} - V_2)^2 / \sum_{l=1}^{n} (V_{y,l} - V_2)^2 \tag{5.67}$$

From the equations above, we know that usually $I_{i,j}/I_{x,i}$ does not equal $I_{y,j}/I_y$. Could they be nearly equal? Of course, if V_2 nearly equals V_1, this happens. If the total I_y equals the total I_x, every $I_{x,i}$ is smaller than the total I_y. As a result, V_1 is always higher than V_2. However, if there exists an $I_{x,i}$ that nearly equals the total I_x, (This usually happens when the probability difference is large for the input), V_1 is nearly equal to V_2 and $I_{i,j}/I_{x,i}$

nearly equals $I_{y,j}/I_y$. Of course, the largest $I_{x,i}$ contributes the most to the output. Thus, the "sum-product" circuit could work correctly.

From Equation 5.64 and 5.65, we know that usually $I_{i,j}/I_{x,i}$ does not equal $I_{y,j}/I_y$. For simplicity, we analyze the equation with only two $V_{y,l}$ first. Let us define $(V_{y1}-V_1)^2 = a$, $(V_{y2}-V_1)^2 = b$, $(V_{y1}-V_2)^2 = c$, and $(V_{y2}-V_2)^2 = d$. If $a/b < c/d$ then $a/(a+b) < c/(c+d)$. Analysis shows the following result:

If $V_1 < V_2$ and $V_{y1} > V_{y2}$ or $V_1 > V_2$ and $V_{y1} < V_{y2}$ then

$$(V_{y1}-V_1)^2/((V_{y1}-V_1)^2+(V_{y2}-V_1)^2) < (V_{y1}-V_2)^2/((V_{y1}-V_2)^2+(V_{y2}-V_2)^2)$$

else

$$(V_{y1}-V_1)^2/((V_{y1}-V_1)^2+(V_{y2}-V_1)^2) > (V_{y1}-V_2)^2/((V_{y1}-V_2)^2+(V_{y2}-V_2)^2)$$

Now, let us consider more than two $V_{y,j}$. We know that if $a1/b1 < a2/b2$ and $a1/c1 < a2/c2$, then $a1/(a1+b1+c1) < a2/(a2+b2+c2)$. As a result, for multiple $I_{y,j}$ inputs, the biggest current $I_{i,j}$ is magnified while the smallest $I_{i,j}$ is decreased if $V_1 > V_2$. If we choose $I_x = I_y = I_u$, then of course $V_1 > V_2$. As a result, the currents with larger probabilities in the product cell are scaled up.

However, if we choose $I_x = I_y = I_z = I_u$, in the case that some of the products are discarded, the total input for the normalization cell is smaller than the total output of the normalization cell, which is I_u. As a result, in this case, the currents with larger probabilities in the normalization cell are scaled down. In conclusion, the quadratic behavior in the product cell and normalization cell can compensate each other to some degree. The quadratic behavior in the normalization cell is more harmful than the quadratic behavior in the product cell because it scales down the large probabilities.

Now, let us analyze the circuit working under strong inversion. For the product cell working under strong inversion, the following equations are true.

$$(V_{y1}-V_2)^2 + \cdot + (V_{yn}-V_2)^2 = p_x[(V_{y1}-V_2)^2 + \cdot + (V_{yn}-V_2)^2] \tag{5.68}$$

$$V_2 = \frac{1}{n}[(V_{y1}+\cdot+V_{yn}) - \sqrt{(V_{y1}+\cdot+V_{yn})^2 - n(V_{y1}^2+\cdot+V_{yn}^2 - p_x[(V_{y1}-V_2)^2+\cdot+(V_{yn}-V_2)^2])}] \tag{5.69}$$

It is very difficult to analyze the circuit working under strong inversion. The $\frac{p'_{yi}}{p_{yi}}$ scale depends not only on p_x, but also on all the $V_{y1}, \cdots, V_{yn}$, and V_1. Even if $p_x, p_{y1}, \cdots, p_{yn}$ are

the same, if different voltages are used, a different result is obtained. For the normalization cell, the situation is much more complex because the ratio of total input/total output is not a simple value such as p_x. Instead, the ratio depends not only on the structure of how the product items are summed, but also on the value of the product items that varies. Only in some cases where no product items are discarded and the total input to a normalization cell equals I_u, no quadratic behavior needs to be considered for the normalization cell.

What is more, the quadratic behavior is more complex than what we have discussed. Although the reference current I_u is very large, a large number of transistors are not working under strong inversion, instead they may be working in the moderate region or in the weak inversion region. For example, if $I_u = 10uA$, $p_{x1} = 0.01$, $I_{x1} = p_{x1} * I_u = 100nA$, some transistors are working in the moderate region or in the weak inversion region.

In conclusion, quadratic behavior is difficult to analyze. It may vary considerably for different analog decoders and different structures. The task is left for simulation to solve.

In order to simulate circuits working under strong inversion and moderate region, the transistor model shown in Equation 5.70 is used to model each transistor. The model shown in Equation 5.70 is proposed by Enz, Krummenacher, and Vittoz [15] and this model is adaptable for transistors working under any region. As a result, using this transistor model, an accurate simulation result can be obtained.

$$I_{DS} = I_S \left[\ln \left(1 + e^{\frac{\kappa(V_G - V_T) - V_S}{2U_T}} \right) \right]^2$$
$$I_S = \frac{2\mu C'_{ox} U_T^2}{\kappa} \frac{W}{L} \tag{5.70}$$

However, the model shown in Equation 5.70 is quite complex and using it to do simulation is too time consuming. It is true that even if the reference current I_u is quite large, some transistors work in the moderate or weak inversion region if the corresponding probabilities for these transistors are small. However, in these cases, the decisions are clear and the largest probabilities are nearly unchanged so that we can treat the cell as an ideal behavioral cell. What we care about is the case that no probability is dominant and in this case the dominant transistors all work in the strong inversion region if I_u is large enough. As a result, a simple model shown in Equation 5.71 is used for transistors working in the strong inversion region. If the decision is clear for a cell, an ideal behavioral model is used to model it. If the decision is not clear, then Equation 5.71 is used for the

dominant transistors. Using this method, the simulation process is sped up significantly while the simulation result is still quite accurate.

$$I_{DS} = \frac{\mu C'_{ox}}{2}\frac{W}{L}(V_G - V_S - V_T)^2 \tag{5.71}$$

The simulation result of the extended Hamming (8,4) decoder using Equation 5.70 is shown in Figure 5.7. It shows that it is more harmful than the mismatch effect, especially at high SNR.

5.6 Transistor Size

From the previous discussion, we know that mismatch and the quadratic behavior are the main nonideal reasons for performance loss. Also, simulation has shown that

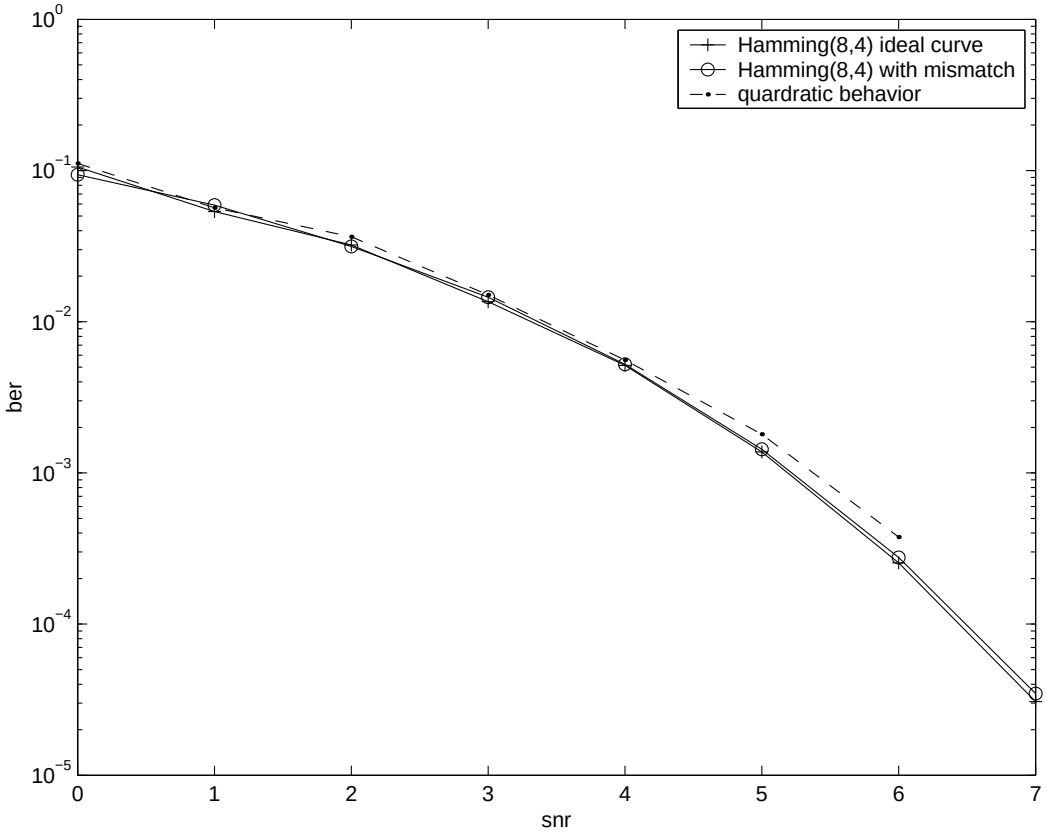

Figure 5.7. The quadratic behavior effect of the tail-biting extended Hamming (8,4) decoder.

the quadratic behavior is much larger than the mismatch effect, especially at high SNR. As a result, we would like to minimize the quadratic behavior first and then minimize the mismatch effect. Also, analysis has shown that the quadratic behavior is mainly dependent on the normalization cell (unless no product term is discarded). As a result, it is better to make the transistors in the current mirror for the X input large and have a small width/length ratio to make it work in the strong inversion region. In contrast, it is better for the transistors of the normalization cell to have a large width/length ratio to make it work in the weak inversion region. The following ratio is recommended: current mirror for the X input W/L=1/4, product core W/L=2, and normalization cell W/L=4.

5.7 One Pole System Simulation

Chapter 3 describes a behavioral level VHDL model for simulation to verify the circuit structure. However, in the behavioral level simulation, a synchronized message passing schedule is assumed. This means that every function node uses one clock cycle to calculate the results. However, a function node may need to use several building blocks to calculate one of its outputs and this number varies for different function nodes. Also, from Chapter 4, we know that a building block is constructed by a product cell and a normalization cell. However, the construction of each building block may use different product cells and normalization cells and the delays of different product cells and normalization cells are different. As a result, for circuit level consideration, the assumption is not true.

For conventional trellis decoding, the message update happens in a forward path and backward path, and in each path the messages are updated sequentially. As a result, if enough time is provided, the circuit stabilizes to the state just as if a synchronized message passing schedule is used. Thus, the circuit level simulation result should be the same as the behavioral level simulation result if the circuit cells are assumed to perform ideally.

For tail-biting trellis decoders and iterative decoding, there are cycles in their factor graph representation. As a result, the circuit delay can affect the performance [1]. Thus, for a more accurate simulation, circuit delay should be considered. Of course, using Spice simulation, the circuit delay is considered. However, it is too time consuming just as

[1]For low-density parity-check codes in which every function node has the same degree and every variable node has the same degree, the circuit delay cannot affect the performance and the behavioral level simulation result is accurate.

mentioned in Chapter 3. To include the circuit delay into the simulation, a good method is to use the *one pole system model* shown in Equation 5.72 to calculate the output for the product cells and normalization cells. In Equation 5.72, $A(\infty)$ is the calculated result that we call stimulation, $A(0)$ is the value of the current output, τ is the transportation delay of the circuit cell provided by a Spice simulation result and t is the time passed since the stimulation is given. Figure 5.8 shows how to use the one pole system model to do simulation. As a result, we need to provide an ideal behavioral model and a delay model for each product cell and normalization cell that are used. When the input is provided to a circuit cell, the stimulation that is needed to be provided to the delay model is calculated first according to the ideal behavioral model. Then, after some delay t, the result of the delay model is passed to another circuit cell. However, the problem is how to choose the delay t. A good method is still using a synchronous schedule and the delay t is chosen as the global clock cycle. Then, if the frequency of the global clock is very high, t is very small. As a result, the messages are updated constantly just as in the analog circuit. Thus, an accurate simulation result can be observed. However, the higher the frequency of the global clock, the more time the simulation uses. As a result, a balance between simulation accuracy and simulation time must be chosen.

$$A(\infty)(1 - e^{\frac{-t}{\tau}}) + A(0)e^{\frac{-t}{\tau}} \tag{5.72}$$

Also, for different $A(\infty)$ and $A(0)$ shown in Equation 5.72, τ may be a little bit different due to the second order effects of a transistor. However, overall the model shown in Equation 5.72 is still accurate enough to provide an accurate simulation result. A more serious impact is the mismatch effect. Due to the mismatch effect, the transistors may not be the ideal desired transistors and the τ provided by the Spice simulation result

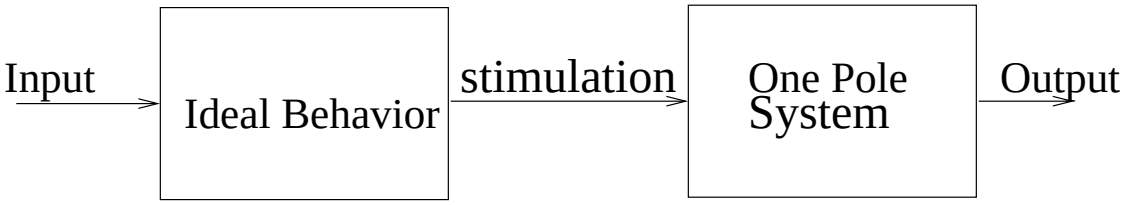

Figure 5.8. One pole system model.

may not be accurate enough. However, to do simulation using the one pole system model and also including the mismatch effect as a gaussian variable is quite complex and too time consuming. Actually, Spice simulation also does not include the mismatch effect. As a result, this method is quite good for both efficiency and accuracy compared with Spice.

The simulation result of the extended Hamming (8,4) decoder shown in Figure 2.11 using this method is shown in Figure 5.9. It shows that the performance is not degraded significantly even if the delay model is used instead of an ideal behavioral model.

5.8 Automatic Circuit Level Simulation

The previous section describes circuit level simulations by using some circuit level models for the circuit cells. Chapter 4 describes that from the factor graph description of a decoder, the automatic synthesis tool can generate the VHDL structural description

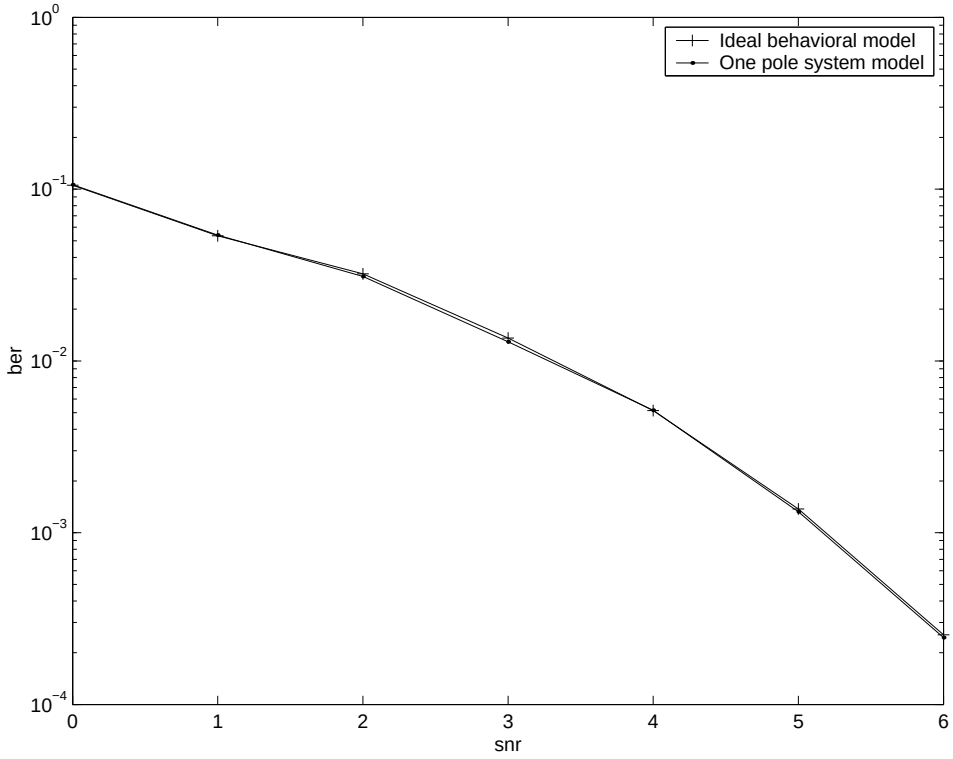

Figure 5.9. Simulation result of the extended Hamming(8,4) decoder using the one pole system model.

of the analog implementation by using the analog circuit cells. Now, if there are circuit level simulation models for the cells in the cell library, then by using these models in the VHDL structural description of the analog decoder, we can do circuit level simulation to obtain circuit level simulation results. As a result, if the one pole system model for the circuit cells are provided, then the circuit level simulation is automated. If the quadratic behavior needs to be simulated, then the transistor level model shown in Equation 5.70 or Equation 5.71 must be provided. If both the one pole system and quadratic behavior need to be simulated, then instead of using the model shown in Figure 5.8, the ideal behavioral model shown in Figure 5.8 should be substituted by the model shown in Equation 5.70 or Equation 5.71.

CHAPTER 6

CASE STUDIES

In this chapter, the extended Hamming (8,4) decoder and the $(16, 11)^2$ product decoder are used to show the result of automatic simulation and synthesis. Also, the advantages and disadvantages of using automatic simulation and synthesis are discussed.

6.1 Hamming (8,4) Decoder

This section uses the Hamming (8,4) decoder as an example. A brief description of the Hamming (8,4) decoder is given first. Then, the high level simulation of the Hamming (8,4) decoder is discussed. The high level simulation result using the automatically generated VHDL file is presented. Also, the Spice simulation result of the automatically generated schematic and layout of the core circuit of the Hamming decoder is presented.

6.1.1 Description

Chapter 2 describes the parity-check matrix and the trellis of the Hamming (8,4) code. Its parity-check matrix is shown in Equation 6.1, and its minimal tail-biting trellis is shown in Figure 6.1. The parity-check matrix H for the Hamming (8,4) code shown in Equation 6.1 has the property that $H \cdot H^T = 0_{4*4}$ so that the parity-check matrix shown in Equation 6.1 is also the generator matrix G for the code.

$$G = H = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \tag{6.1}$$

In order to do the automatic simulation and synthesis of the Hamming decoder. The factor graph description of the Hamming (8,4) code is needed. For the Hamming (8,4) code, there are four trellis sections. Each trellis section describes the valid configurations between the variables, i.e. the current state s_r, the next state $s_{(r+1)mod4}$ (mod is used because the trellis shown in Figure 6.1 is a tail-biting trellis), the information bit u_r,

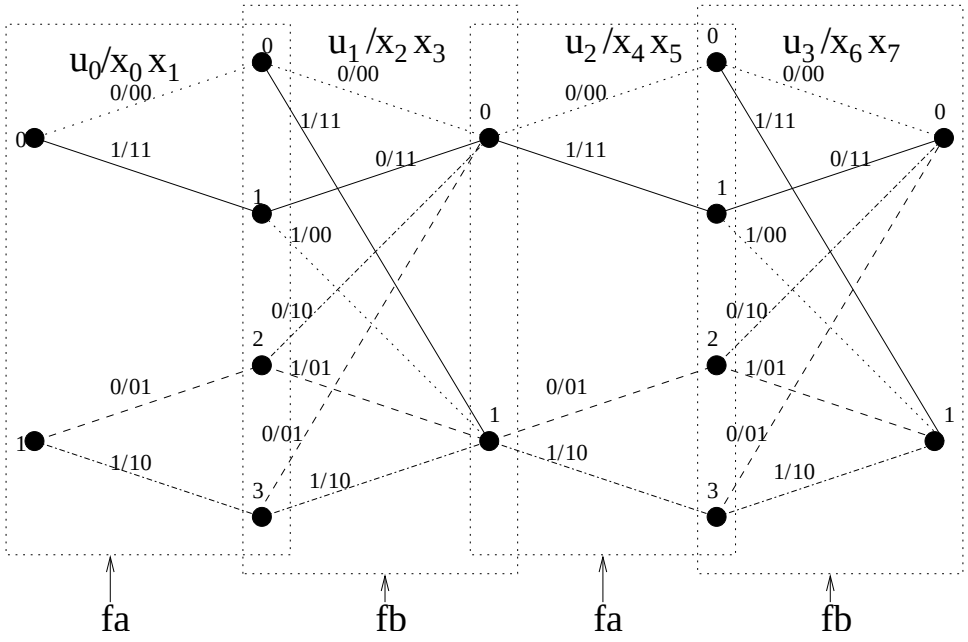

Figure 6.1. Trellis representation for the extended Hamming (8,4,4) code.

and the encoded outputs x_{2r}, x_{2r+1}. As a result, each trellis section can be viewed as a function node connected with five adjacent variables s_r, $s_{(r+1)mod4}$, u_r, x_{2r}, x_{2r+1} in the factor graph in which the function node is an indication function defined by the valid configurations shown in the trellis section. Although there are four trellis sections, there are only two types of trellis sections. As a result, there are only two types of function nodes, as shown in the factor graph representation in Figure 6.2. In the factor graph, the state variable node s_r is not visible and it is shown in a double circle. When doing decoding, the information bit u_r only provides a unity probability distribution to its adjacent function node. As a result, the output of u_r can be ignored in doing simulation and synthesis. Also, we do not need to know the value of the message going to variable node x_r. Thus, only the messages going out of variable node x_r need to be calculated. For convenience, we can combine the messages coming out of variable nodes x_{2r} and x_{2r+1}. Now, the factor graph shown in Figure 6.2 is simplified to Figure 6.3. The function node now only has four adjacent variables and the messages coming out of node x_{2r} and x_{2r+1} is provided by the combined message coming out of node γ_r.

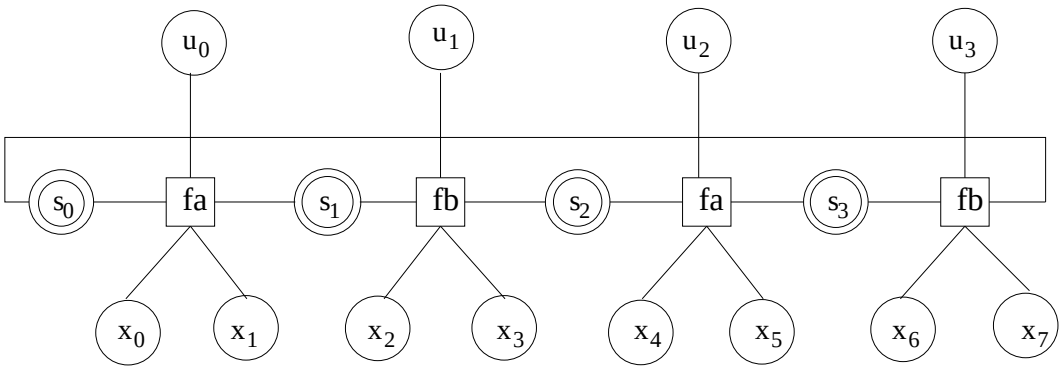

Figure 6.2. Factor graph representation of the Hamming (8,4) decoder.

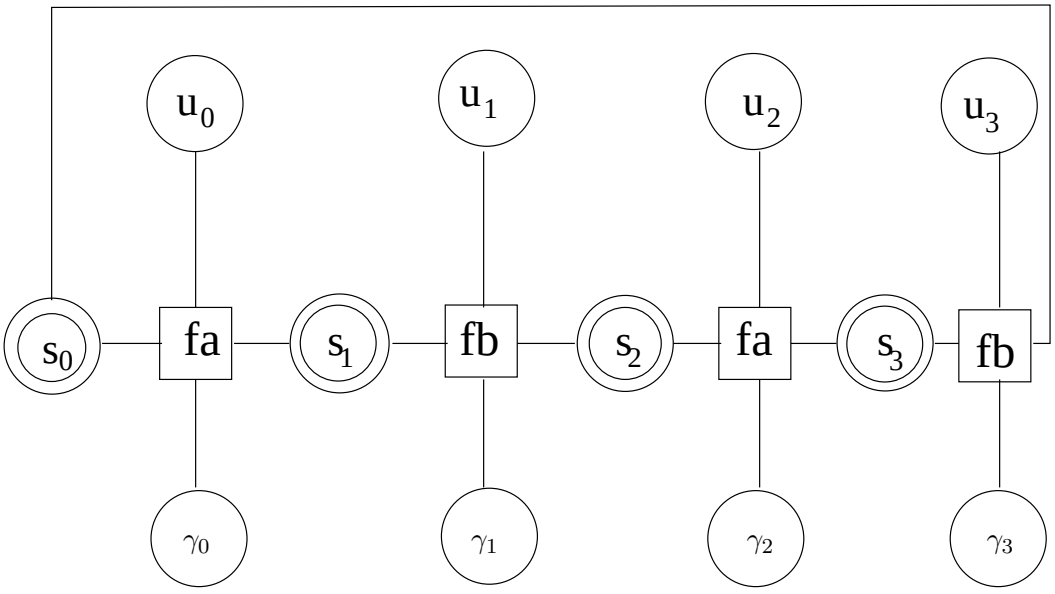

Figure 6.3. Another factor graph representation of the Hamming (8,4) decoder.

6.1.2 High-Level Simulation

The structure of the Hamming (8,4) decoder is represented by its factor graph shown in Figure 6.3. The messages passed on one section of the factor graph are shown in Figure 6.4. For the state variable node s_r, there are messages passing in both the forward and backward direction. The messages passed in the forward direction are represented by α_r while the messages passed in the backward direction are represented by β_r. In order to calculate message α_{r+1}, α_r and γ_r need to be used as shown in Equation 6.2. In order to calculate β_r, β_{r+1} and γ_r need to be used as shown in Equation 6.3. In order to calculate the messages going to variable node u_r, α_r, γ_r, and β_{r+1} all need to be used as shown in Equation 6.4. However, notice that the branches that enter state $r+1$ always correspond to a certain u. In function fa, the branches that enter next state 0 and 2 always correspond to $u = 0$ while the branches that enter next state 1 and 3 always correspond to $u = 1$. In function fa, the branches that enter next state 0 always correspond to $u = 0$ while the branches that enter next state 1 always correspond to $u = 1$. As a result, Equation 6.4 can be simplified to Equation 6.5. As a result, the structure of the Hamming (8,4) decoder that we used in [69] is shown in Figure 6.5. The $bit - pair$ combines the messages provided by x_{2r} and x_{2r+1} shown in Figure 6.2 to provide the messages of γ_r shown in Figure 6.4. The core of the structure is derived from the factor graph shown in Figure 6.3. For the core of the structure shown in Figure 6.5, each block has two message input and one message output. As a result, each block can be built by using a product cell and a normalization cell. The circuit for the first B_2 block is shown in Figure 6.6.

$$\alpha_{r+1}(j) = \sum_{states\, i} \alpha_r(i)\gamma_r(i,j) \tag{6.2}$$

$$\beta_r(i) = \sum_{states\, j} \beta_{r+1}(j)\gamma_r(i,j) \tag{6.3}$$

$$u_r(u) \quad = \quad \sum_{(i,j)\in A(u)} \alpha_r(i)\gamma_r(i,j)\beta_{r+1}(j) \tag{6.4}$$

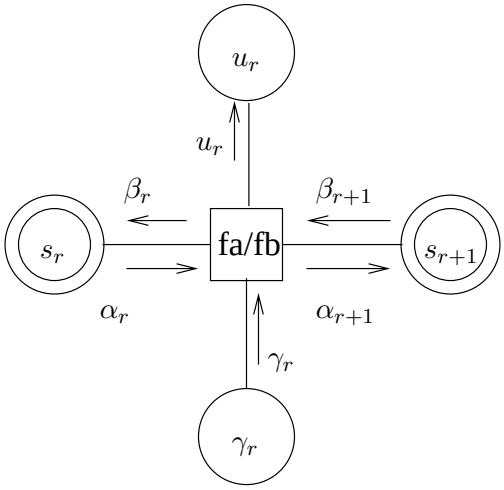

Figure 6.4. A detailed view of the messages for a single trellis section of the Hamming (8,4) decoder.

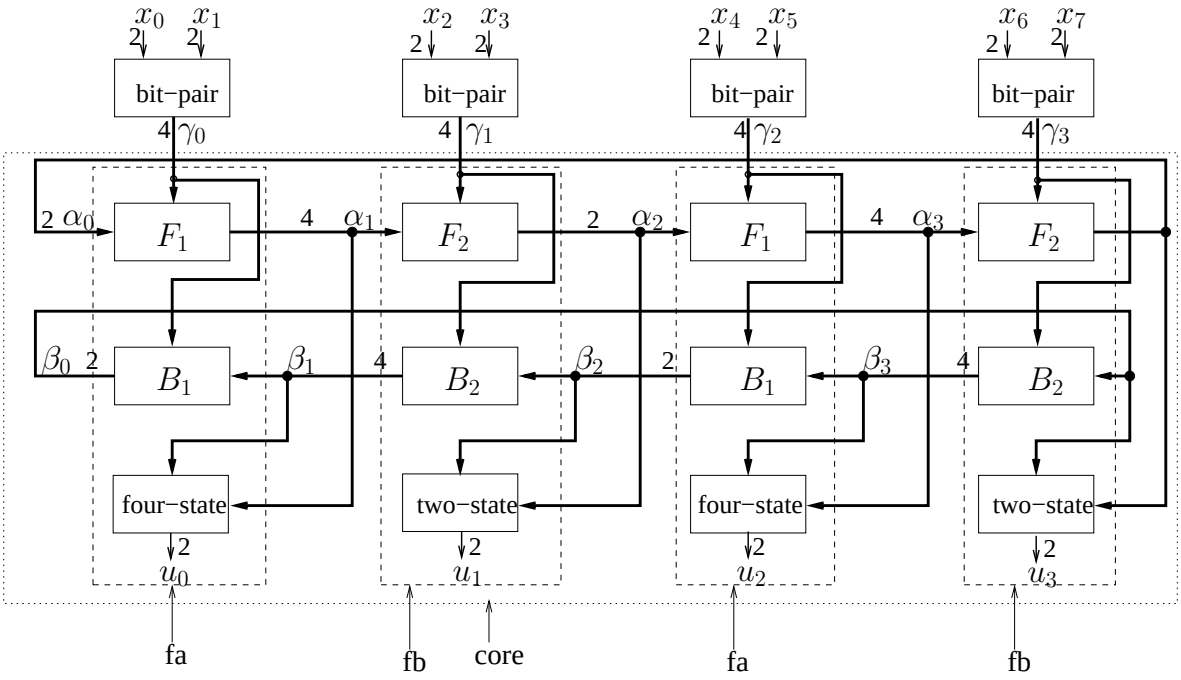

Figure 6.5. Block diagram of the (8,4) Hamming decoder.

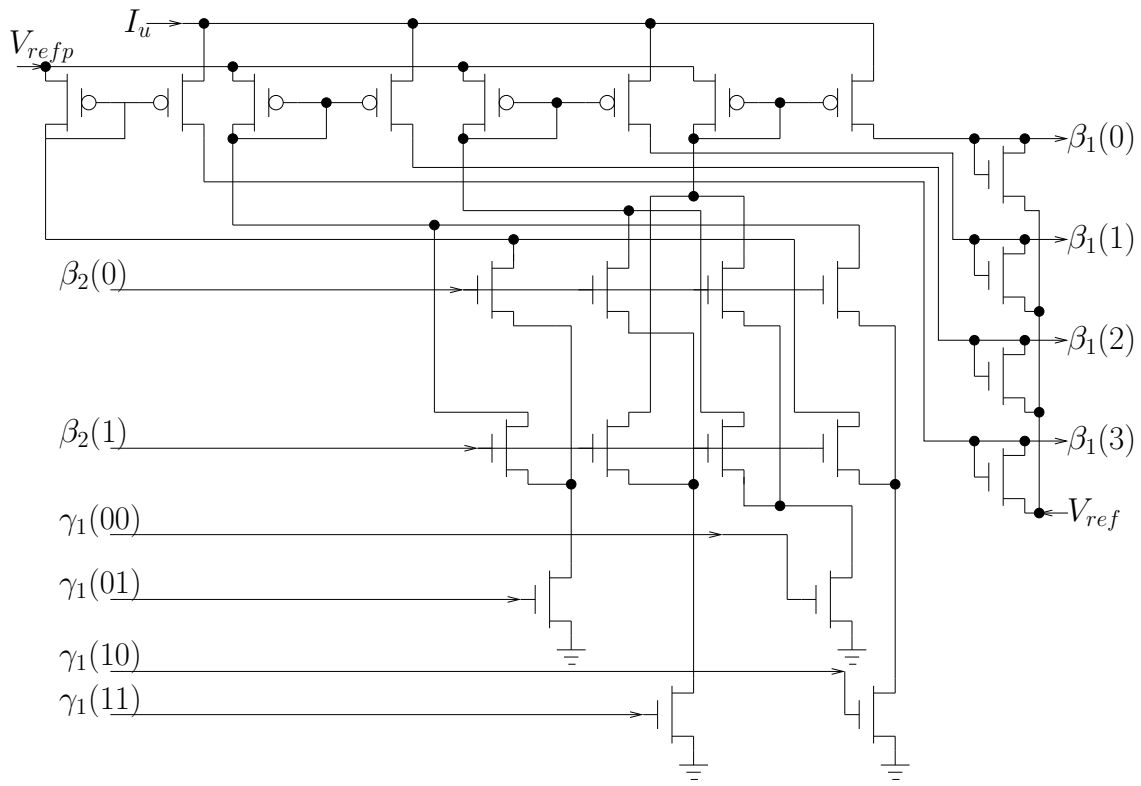

Figure 6.6. Circuit for B_2.

$$
\begin{aligned}
u_r(u) &= \sum_{(i,j)\in A(u)} \alpha_r(i)\gamma_r(i,j)\beta_{r+1}(j) \\
&= \sum_{(j)\in A(u)} \beta_{r+1}(i+1) \sum_{i} \alpha_r(i)\gamma_r(i,j) \\
&= \sum_{(j)\in A(u)} \alpha_{r+1}(j)\beta_{r+1}(j) \qquad (6.5)
\end{aligned}
$$

In order to verify whether the structure shown in Figure 6.5 is correct or not, high-level VHDL simulation is used to get the bit error rate curve shown in Figure 4.27. The simulation verifies that the structure is correct. Moreover, in doing the high-level simulation, it is found that for the tail-biting trellis decoder, the initial condition when a new codeword arrives is the end condition when the previous codeword is decoded. For the factor graph simulation, the initial condition is often assumed to be unit messages on every edge, and it is proven that factor graphs with a single cycle are guaranteed to converge with a reasonable initial condition such as unit message distribution. If the initial condition when a new codeword arrives is the end condition when the previous codeword is decoded, then a long time may be needed for the decoder to converge to the correct result in a high SNR condition. In extreme cases such as no noise, the initial condition when a new codeword arrives may be so unreasonable that the decoder may not work. As a result, a reset circuit is proposed. When a codeword has been decoded, the messages along the cycle are reset to a unit distribution. Using this method, the decoder is guaranteed to work for any SNR condition, and it is sped up by the reset method in high SNR conditions as shown in Figure 6.7. In Figure 6.7, the flooding message passing schedule and a global synchronous clock are used. Four time units means that four clock cycle have passed since the arrival of the input probabilities.

6.1.3 Automatic High-Level Simulation

From the discussion above, we can see that the Hamming (8,4) decoder is completely described by its factor graph representation. Also, the factor graph description is quite powerful. For example, the indication function for function node fa can be expressed as $(s_r, u_r, \gamma_r, s_{r+1}) = (0,0,00,0;0,1,11,1;1,0,01,2;1,1,10,3)$ in which every string separated with the semi-colon shows a valid configuration. This simple description describes

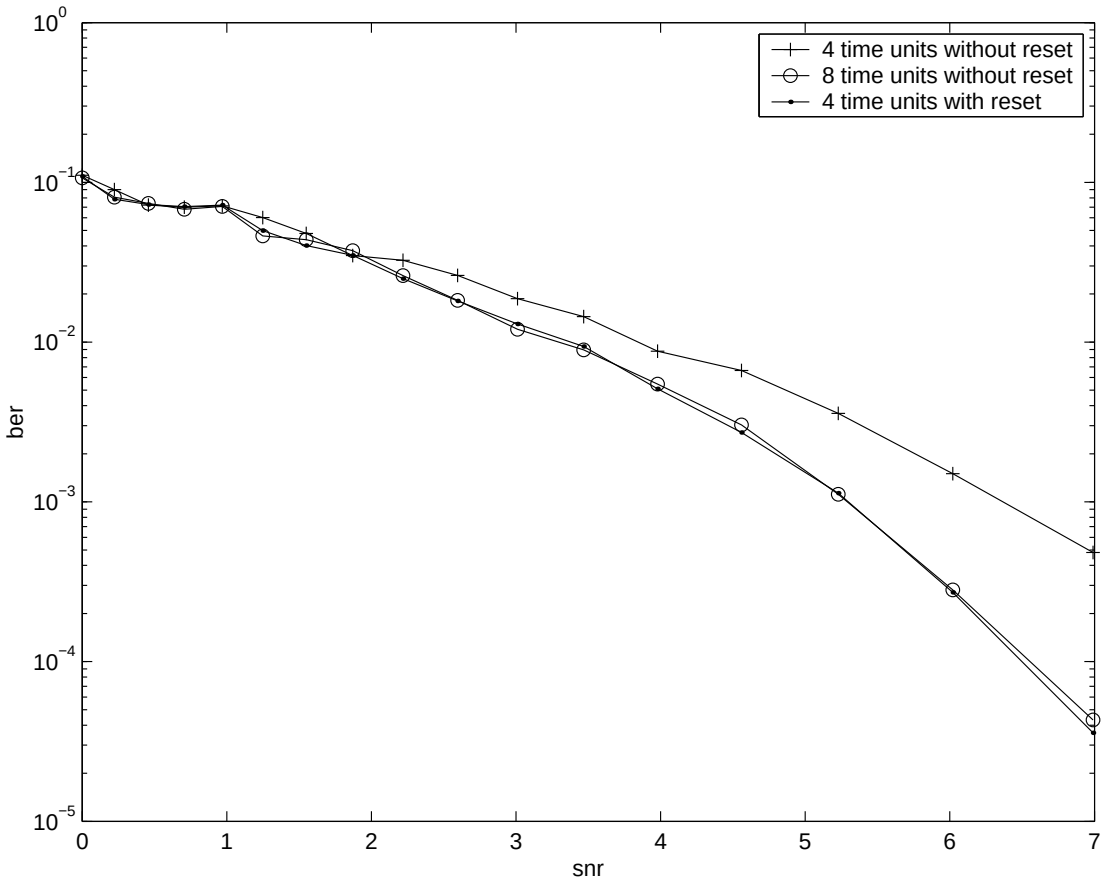

Figure 6.7. Simulation result of the extended Hamming(8,4) decoder using and not using a reset circuit.

all the operations for the function node, which is much more complex using VHDL to describe. Using automatic simulation, the VHDL simulation file is generated by the tool using the factor graph as input. The high-level VHDL simulation result for the Hamming (8,4) decoder using the automatically generated VHDL file is shown in Figure 6.8. The simulation result using the automatically generated VHDL file is the same as using the hand written VHDL file. However, writing the VHDL file by hand may take 1 day while writing the factor graph description takes only 1 hour and does not require the knowledge of VHDL, which is advantageous for a digital communication expert. Of course, the simulation process is sped up.

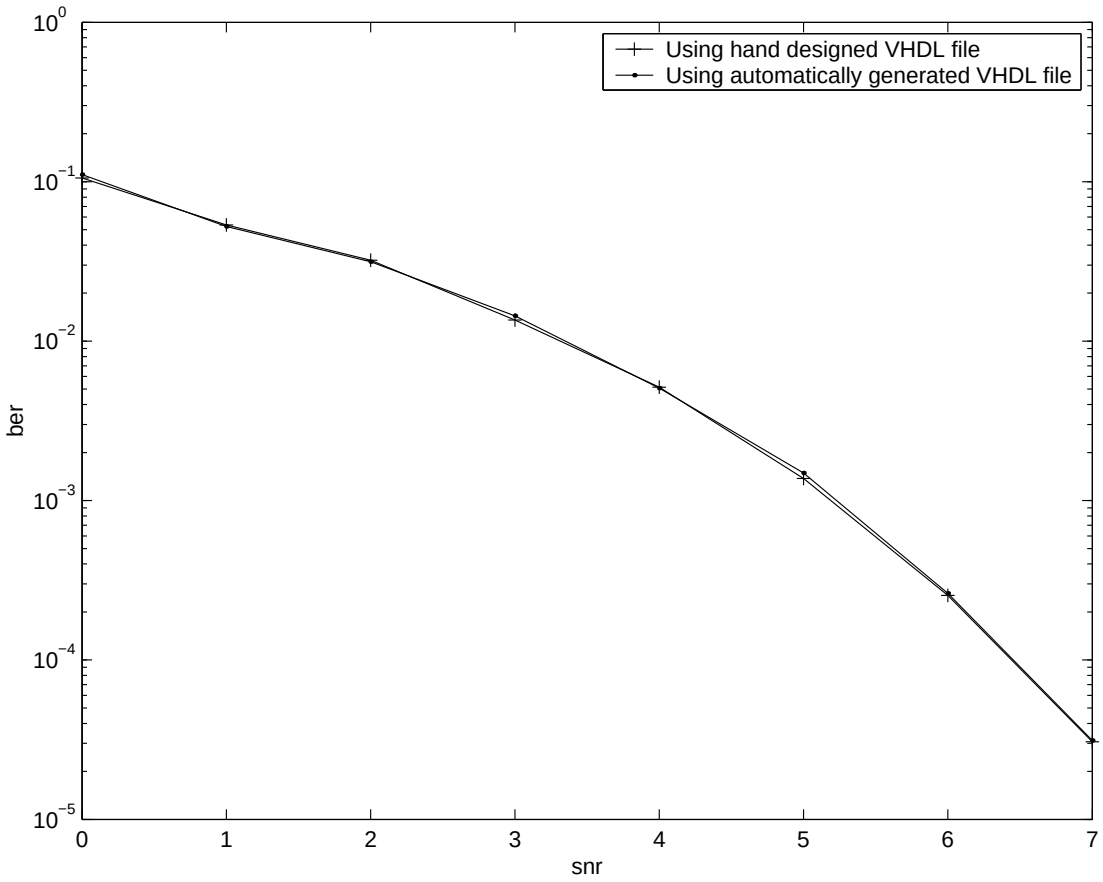

Figure 6.8. Simulation result of the extended Hamming(8,4) decoder using the automatically generated VHDL file.

6.1.4 Automatically Generated Circuit

The previous section describes how to derive the circuit for the Hamming (8,4) decoder from its factor graph representation. Drawing the circuit by hand is time consuming and error prone. It takes about 1 month for a person to draw the schematic and layout for the core of the Hamming (8,4) decoder from its factor graph representation, and we even made a mistake when drawing the circuit for our first Hamming (8,4) decoder chip [69]. Actually, all the work can be simplified using the automatic synthesis tool and cell library presented in this thesis. The automatic synthesis tool also uses the factor graph description of the decoder, which is quite simple. With the aid of other commercial tools, the schematic and layout of the core of the Hamming (8,4) decoder are generated automatically. The layout of the automatically generated core circuit of the Hamming (8,4) decoder is show in Figure 6.9. The structure of the automatically generated circuit is similar to the structure of the hand derived circuit shown in Figure 6.5 except that the function nodes fa and fb are directly constructed by the product cells and normalization cells without partitioning it into the forward path cell, backward path cell, and two-state or four-state cells. The Spectre simulation result of the automatically generated schematic and layout is shown in Figure 6.10. In the simulation, the voltage sources for the product cells and normalization cell is chosen as follows: $Vdd = 5V$, $Vu = 4.1V$, $Vrefp = 3.9V$, $Vrefn = 0.5V$, and $Vdummy = 3V$. Vu is chosen to be $4.1V$ to give the reference current $Iu = 10nA$ for the circuit to work in the subthreshold region. $Vrefp$ is chosen to be a little lower than Vu so that the normalization cell can work properly. $Vdummy$ is the voltage to be connected to the drain of the dummy transistors used in the product cells. Since $0.9V$ is needed for the gate-source voltage of the PMOS transistors to provide the drain current of $10nA$ that equals Iu, the drain voltage for the NMOS transistor that is on the correct path is likely to be $Vrefp - 0.9 = 3V$. Thus, choosing $Vdummy = 3V$ can minimize the channel length modulation effect. The input to the circuit is chosen to correspond to the probabilities of all the $x_i = 0, i = 0, \cdots, 7$ to vary between 0.4 and 0.6 periodically with a $2ms$ period. Corresponding to this input, the decoded codeword should change between 0000 and 1010 periodically. The simulation results for the probability of u_0 to be 0, u_0_0, and to be 1, u_0_1, are shown in Figure 6.10. It shows that both the automatically generated schematic and layout work correctly. Using the automatic synthesis tool, it only takes about 1 to 2 hours to generate the schematic and layout for the core of the Hamming (8,4) decoder including writing the factor graph description files and using the

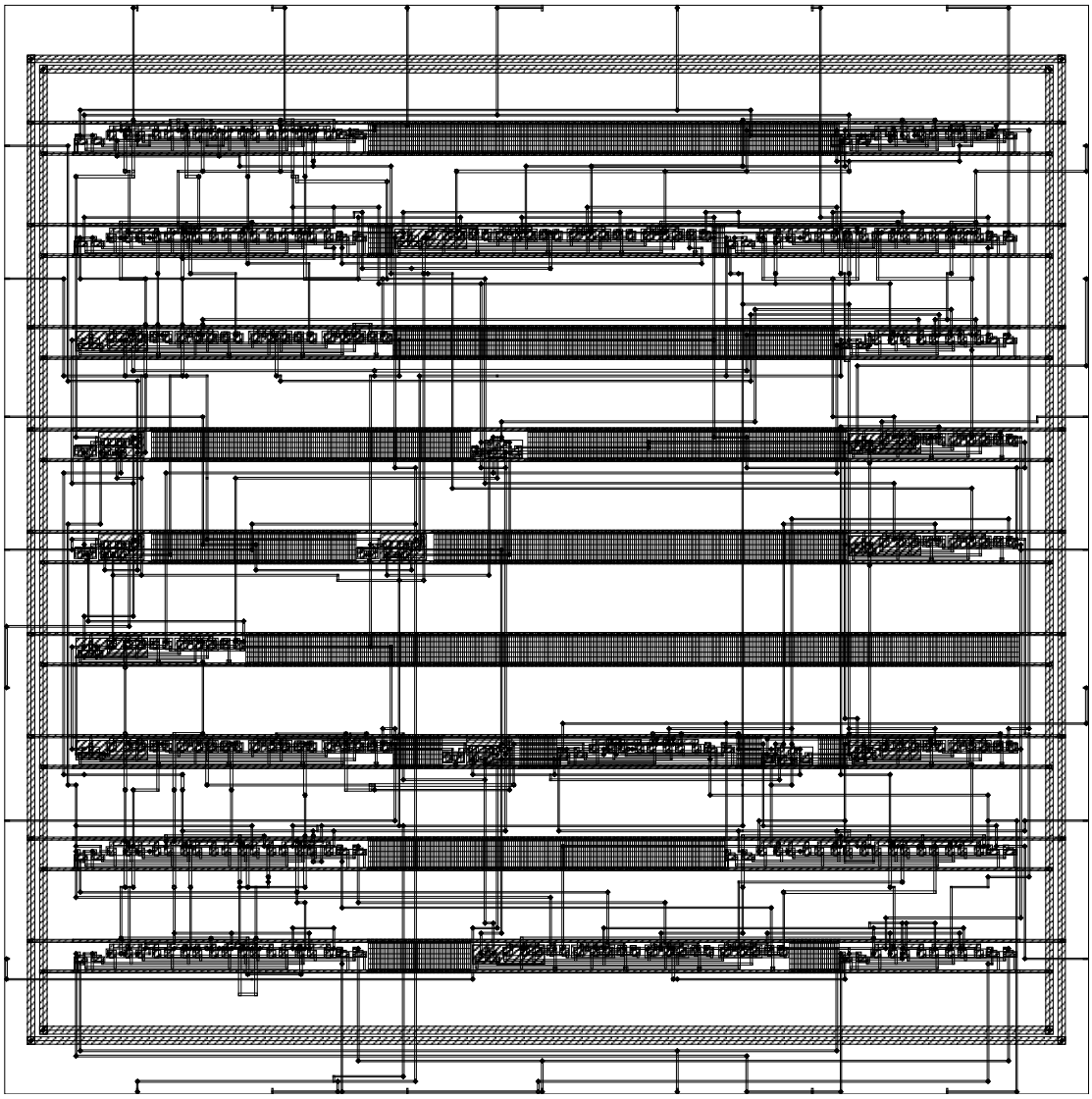

Figure 6.9. Layout of the core of the Hamming (8,4) decoder.

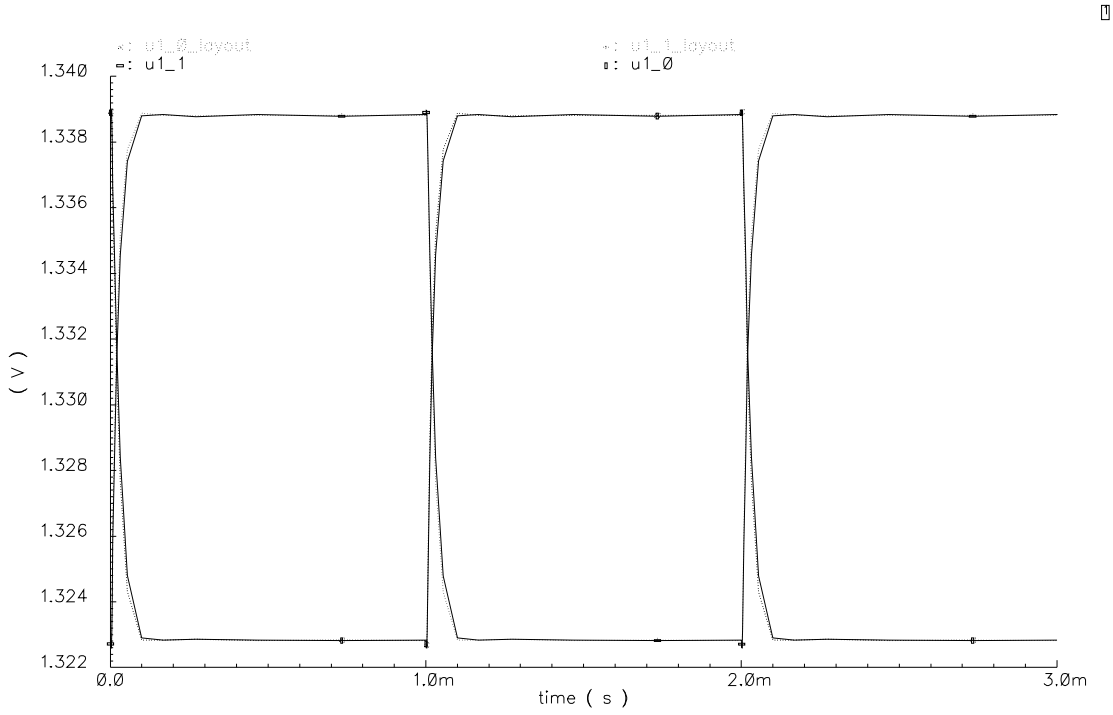

Figure 6.10. Spectre simulation result of the automatically generated schematic and layout of the Hamming (8,4) decoder.

tools. Compared with the hand design, substantial time is saved.

However, there are also disadvantages of the automatically generated circuit. For the cell library we used, the product cells use voltage input. Thus, the connections between blocks are voltage connections as shown in Figure 4.13. The product cell needs both X and Y direction inputs, which have different voltage potentials. Sometimes the output of one normalization cell needs to be provided to the X input of a product cell and Y input of another product cell. In this circumstance, the normalization cell needs to provide two sets of outputs, one for the X input of the product cells and another set for the Y input of the product cells. However, it is quite difficult to decide in advance which product cells use the output of one particular instance of a normalization cell and whether the product cells need X or Y output from this normalization cell, especially for a larger decoder. As a result, in the current tool, for the intermediate result provided by a block, both the X and Y outputs are generated. Thus, sometimes a little bit of redundant circuitry is generated. For example, for the Hamming (8,4) decoder shown in Figure 6.5, the γ value is always provided as an X direction input to the blocks. α and β values are always provided as Y direction input to the next stage along the forward path and backward path for the speed consideration discussed in Section 4.5.1. However, the two-state and four-state block need one X input and one Y input. Thus, one of the α and β values should provide X input to the two-state or four-state block. For the hand design, we choose the forward path to provide α as both X output and Y output for other blocks to use while only providing β as Y output along the backward path. However, for the automatically generated circuit, X and Y outputs are provided for both the α and β values, making the core of the Hamming (8,4) decoder uses 16 more transistors than the hand design. Another disadvantage is that the automatically generated layout is much larger than the hand design. The automatically generated circuit for the Hamming (8,4) decoder is about $600um * 600um$, which is much larger than the hand design that is only about $200um * 300um$. Even if the automatically generated layout can be made more tight compared with the current design, the automatically generated layout will likely still be larger than the hand design.

6.2 $(16, 11)^2$ Product Decoder

This section use the $(16, 11)^2$ product decoder as an example [70]. A brief description of the $(16, 11)^2$ product decoder is given first. Then, the high level simulation of

the $(16, 11)^2$ product decoder is discussed. The high level simulation result using the automatically generated VHDL file is presented. Also, the Spice simulation result of the automatically generated schematic and layout of the core circuit of the $(16, 11)^2$ product decoder is also presented.

6.2.1 Description

The structure of a product decoder has been described briefly in Section 2.10. The product decoder that is discussed here is a $(16, 11)^2$ product decoder constructed by the Hamming (16,11) trellis decoder as its component decoder. The generator matrix for the Hamming (16,11) decoder that we are using is shown in Equation 6.6.

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \tag{6.6}$$

The compact trellis for this code may be built using the "squaring construction." The squaring construction begins with simple trellises for partitions of the Hamming (16,11) code, and then attaches them to create the completed code trellis. The simple trellis used for the partition of the Hamming (16,11) code is shown in Figure 6.11 and it can be viewed as the disjoint subsets shown in Equation 6.7. Using these subsets, the compact trellis for the Hamming (16,11) code is shown as Figure 6.12. Then, 16 Hamming (16,11) decoders are used to construct 16 row decoders and 16 Hamming (16,11) decoders are also used to construct 16 column decoders and the full version product decoder is constructed by using 256 equal gates shown in Figure 2.15 to connect the 16 row decoders and column decoders shown in Figure 6.13. If no parity-on-parity bits are used and the parity bits are not checked by other component decoders, only 11 row decoders and 11 column decoders are needed as shown in Figure 6.14. Now, only the information bits are checked by both a row decoder and a column decoder and an equal gate is only needed for every information bit. The parity bits are only used by one component decoder and the channel information

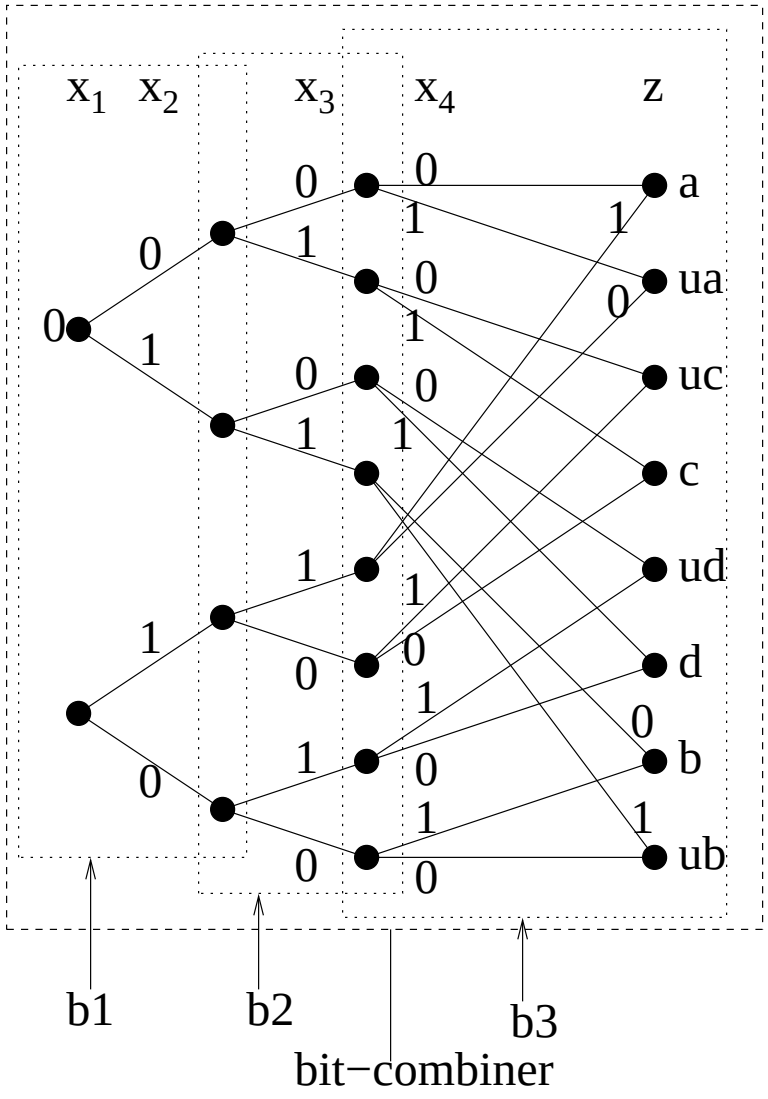

Figure 6.11. Simple trellis for the Hamming (16,11) code.

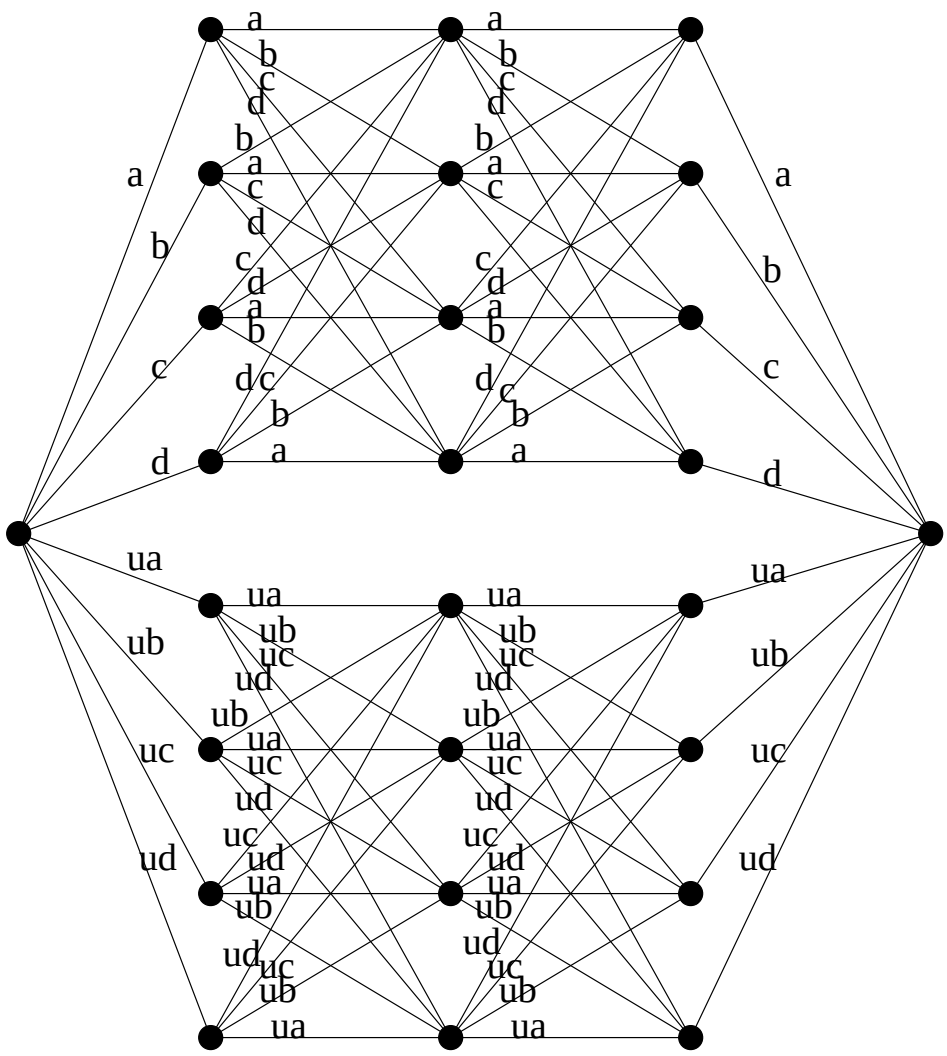

Figure 6.12. Trellis for the Hamming (16,11) code.

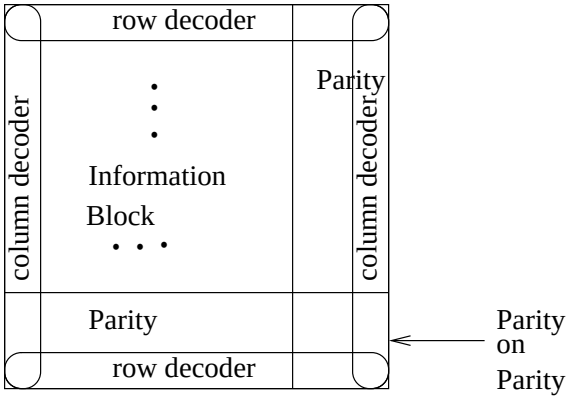

Figure 6.13. "Full version" product decoder.

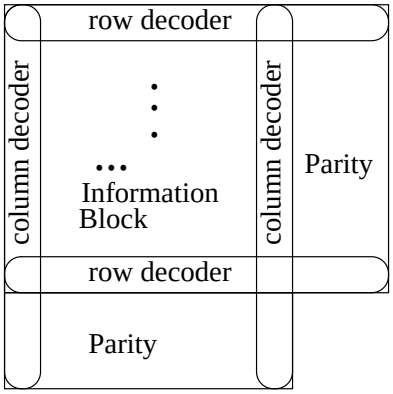

Figure 6.14. "Punctured version" product decoder.

for the parity bits are passed directly to the component decoder. This kind of product decoder is called a "punctured version" as shown in Figure 6.14 while the product decoder shown in Figure 6.13 is called a "full version."

$$
\begin{aligned}
a &= \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \\
b &= \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \\
c &= \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} \\
d &= \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \\
ua &= \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \\
ub &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix} \\
uc &= \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix} \\
ud &= \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix}
\end{aligned} \tag{6.7}
$$

6.2.2 High-Level Simulation

Initially, we believed that the performance of the "punctured version" should be good enough and the "full version" would not improve the performance. Therefore, we chose the "punctured version" product decoder. A high-level VHDL simulation is used to verify the structure of the "punctured version" and its bit error rate curve is shown in Figure 6.15. However, its performance is not as good as expected. High-level VHDL simulation of the "full version" shows that the "full version" product code has a much better performance than the "punctured version", especially at high SNR, as shown in Figure 6.15. A theoretical analysis also verifies that the "full version" should have a much better performance than the "punctured version." Therefore, we decided to build a "full version" product decoder instead of a "punctured version." The high-level simulation of both the "punctured version" and the "full version" product decoder also shows that as the number of iterations increases, the bit error rate decreases. However, after six iterations, there is no significant bit error rate decrease with the increase of iterations.

For the product decoder, there are many cycles. As a result, if no reset circuit is

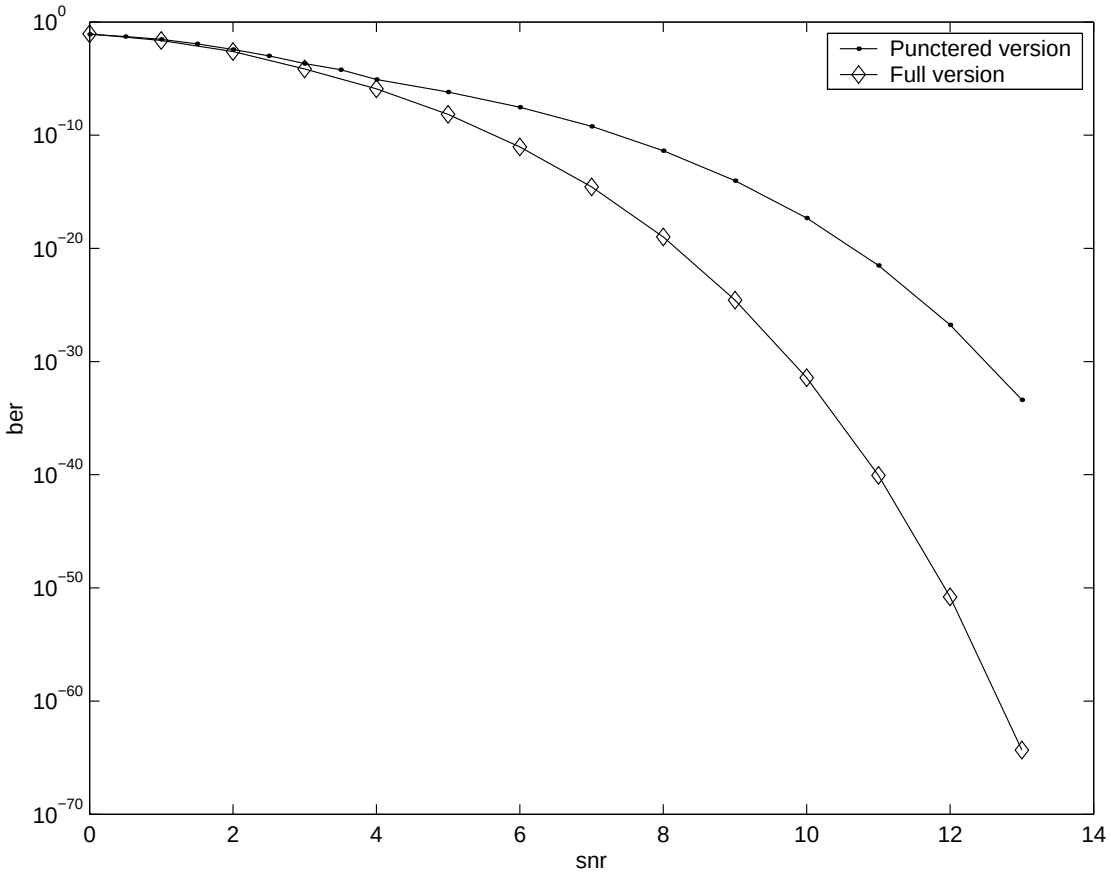

Figure 6.15. Simulation result of the $(16, 11)^2$ product decoder.

used, the initial condition for codeword decoding is the end condition for the previous codeword decoding. Thus, when a new codeword arrives, the component decoder is provided with extrinsic information generated by the previous codeword. This is very harmful. Simulation result of the "punctured version" $(16, 11)^2$ product decoder shown in Figure 6.16 shows that the bit error rate is lower if reset is used compared with the case no reset is used. In Figure 6.16, the flooding message passing schedule and a global synchronous clock is used, and 40 clock cycles are used to do the simulation. Forty clock cycles is the time needed for four iterations of the component decoder. If 58 clock cycles that corresponds to six iterations, are used to do the simulation, the simulation result shown in Figure 6.17 shows that there are no significant performance increase if reset is used. However, simulation of the "full version" $(16, 11)^2$ product decoder shows that even if SNR is as small as 0, the decoder cannot work if no reset is used. Even when SNR=0, for the "full version" $(16, 11)^2$ product decoder, the extrinsic information provided to the equal gate can reach an absolute value such as the probability of being '0' is 1 while the probability of being '1' is 0 after a codeword is decoded. Then, when the next codeword arrives, the decoder may not make a correct decision because the probabilities along the cycles are already fixed to absolute 0's and 1's. Actually, all the data points shown in Figure 6.15 are obtained by using reset in the decoder.

Also, in doing high level simulation, it is found that for the analog iterative decoders, a component decoder constantly provides extrinsic information to other component decoders even when the component decoder that provides the extrinsic information has not stabilized and finished decoding. However, even if the component decoder has not stablized and finished decoding, the generated extrinsic information is based on part of the component decoder's codewords. We can call this extrinsic information "partial extrinsic information" and it also helps the whole decoder to converge to the final result. Thus, providing the extrinsic information constantly should not degrade the performance. Also, if the extrinsic information is provided to other component decoders only when the component decoder has finished decoding, then the extrinsic information generated by the previous iteration needs to be stored and there are no efficient analog circuits that can store the probabilities represented by currents. Actually, simulation of both the "punctured version" and "full version" $(16, 11)^2$ decoder shown in Figure 6.18 shows that providing the extrinsic information constantly improves the performance slightly. All in all, simulation verifies the utility of the factor graph simulation methodology that uses

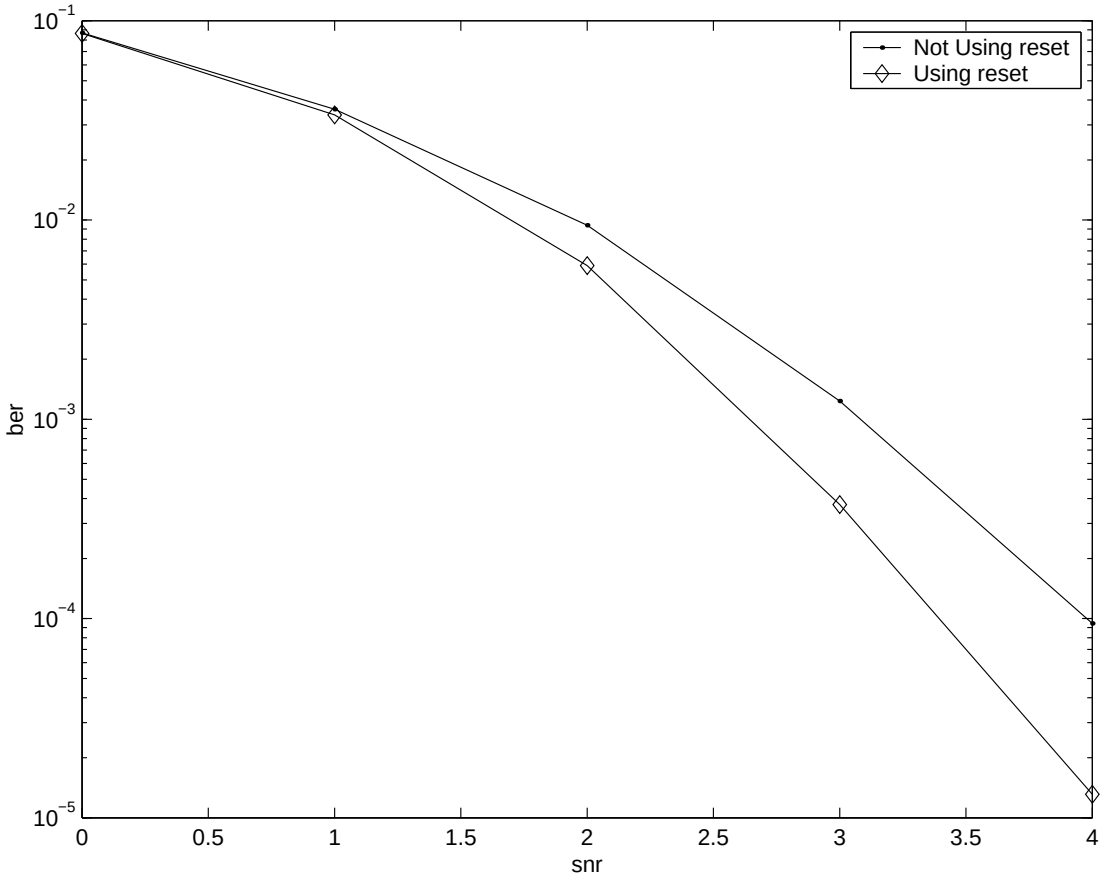

Figure 6.16. Comparsion of using and not using reset (40 time units are used).

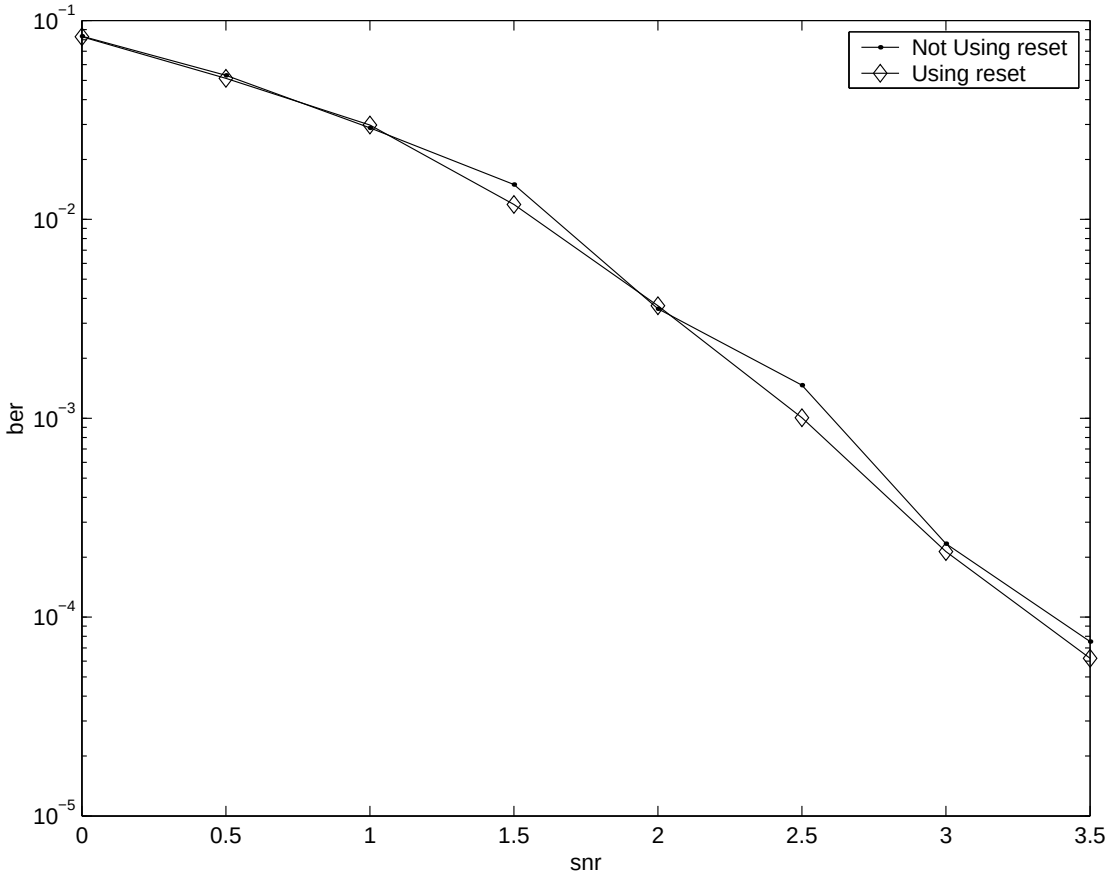

Figure 6.17. Comparsion of using and not using reset (58 time units are used).

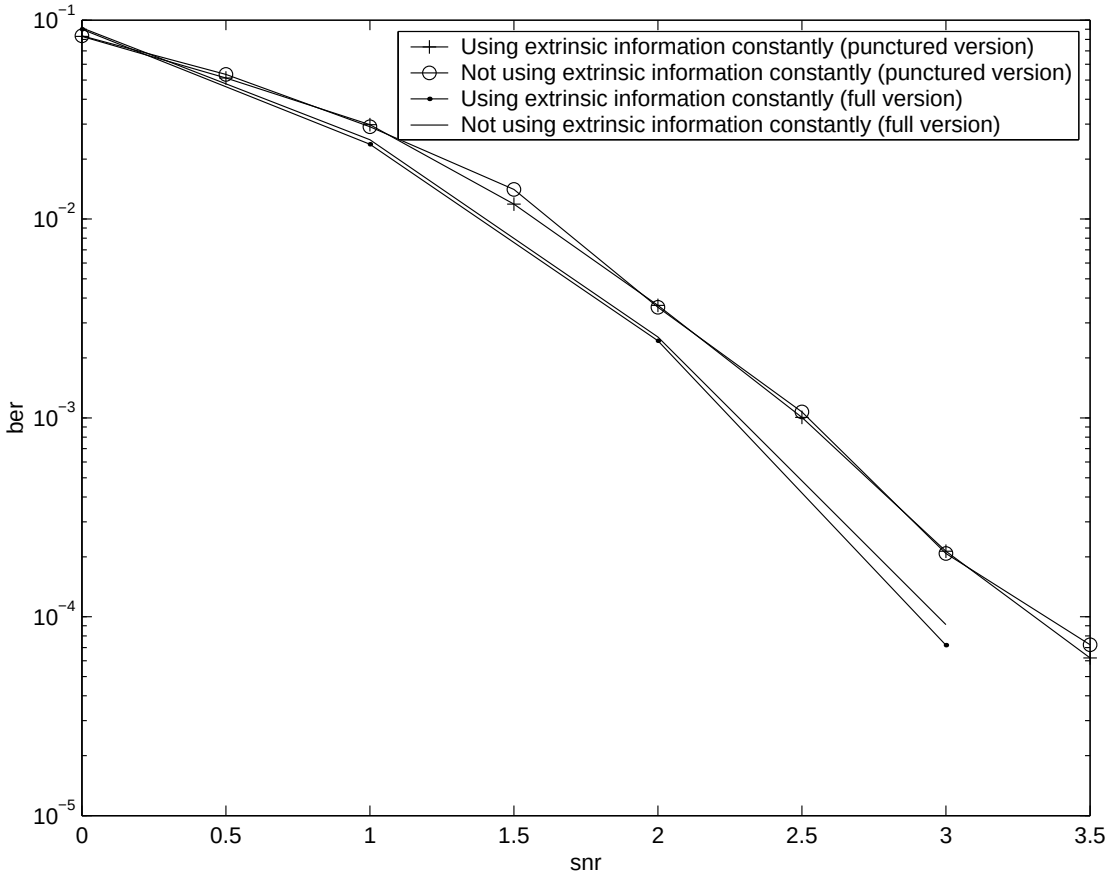

Figure 6.18. Using and not using extrinsic information constantly.

flooding message passing schedule and uniform initial probability distribution. For the implementation of the $(16, 11)^2$ product decoder, we do not need to do reset on all the circuits that are in some cycles. Actually, we can reset only the extrinsic information provided by the component decoders to uniform distribution for a time long enough so that the component decoders are not affected by the extrinsic information generated with the previous codeword in the first iteration. Thus, the end condition of the previous codeword cannot affect the decoding of the current codeword.

Also, even if a high-level VHDL simulation is used to get the bit error rate curve, the simulation for the product decoder is quite time consuming, especially at high SNR. The simulation of the "punctured version" product decoder at SNR=3 takes about 10 hours on a Pentium 4 machine and about 1 week is needed for the simulation at SNR=4. Fortunately, there is an *importance sampling* method recently proposed by Ferrari and Bellini [16]. The importance sampling method is quite efficient for high SNR simulations compared with the traditional Monte-Carlo simulation method. Our simulations of the product decoder show that importance sampling is not as efficient as Monte-Carlo simulation for SNR<=3. However, when SNR>3, importance sampling becomes more efficient compared with Monte-Carlo simulations as SNR increases. Actually, the simulation result for SNR>3 shown in Figure 6.15 is obtained using importance sampling and it takes about 18 hours for importance sampling to get all the data points of the "full version" product decoder for SNR>3 shown in Figure 6.15 on a Pentium 4 machine.

6.2.3 Automatic High Level Simulation

It takes about 1 week to write the VHDL simulation file for the product decoder. Using the automatic simulation tool, it takes only about 1 hour to write the factor graph description file and then no more than 1 minute for the automatic simulation tool to generate the VHDL file. The simulation result for the "full version" product decoder using the automatically generated VHDL file is shown in Figure 6.19. It shows that the VHDL file is generated correctly and the simulation result is correct.

However, for the importance sampling method, the simulation environment should provide codewords according to the error centers. However, the error centers are decided by the code and it is quite difficult to generate the error centers automatically. At the current time, the automatic high level simulation can provide only the simulation environment for Monte-Carlo simulation. Automatic simulation for importance sampling is left for future work.

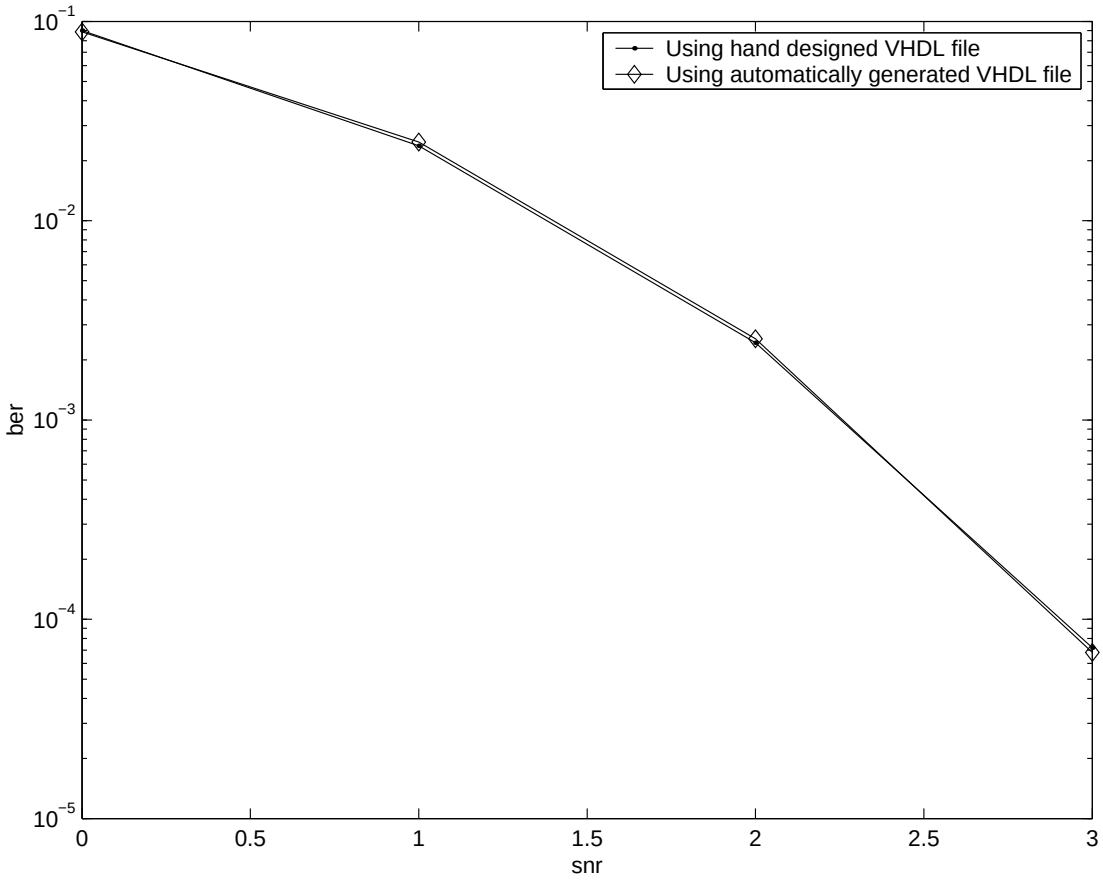

Figure 6.19. Simulation result of the "full version" $(16, 11)^2$ decoder using the automatically generated VHDL file.

6.2.4 Automatically Generated Circuit

The circuit for the product decoder is quite large. Even drawing the schematic of the product decoder takes about 1 week. Using the automatic synthesis tool, only about 1 hour is needed to write the factor graph description files and use the tools to get the schematic. There is a little bit of redundant circuit compared with the hand design as stated in Section 6.1.4. The Spectre simulation result for the automatically generated circuit is shown in Figure 6.20 using the same input condition as the simulation of the Hamming (8,4) decoder. The simulation result of probabilities of the upper left corner bit to be 0, $u\_1\_1\_0$ and to be 1, $u\_1\_1\_1$ shown in Figure 6.20 shows that the generated circuit is correct.

However, automatic generation of the layout is quite difficult. Even using an area of $1.0cm * 1.4cm$, it is still quite difficult for Silicon Ensemble to do the routing between the cells, resulting in a large number of geometry violations and then generating a large number of DRC errors. Also, the wire connections between cells are usually quite long,

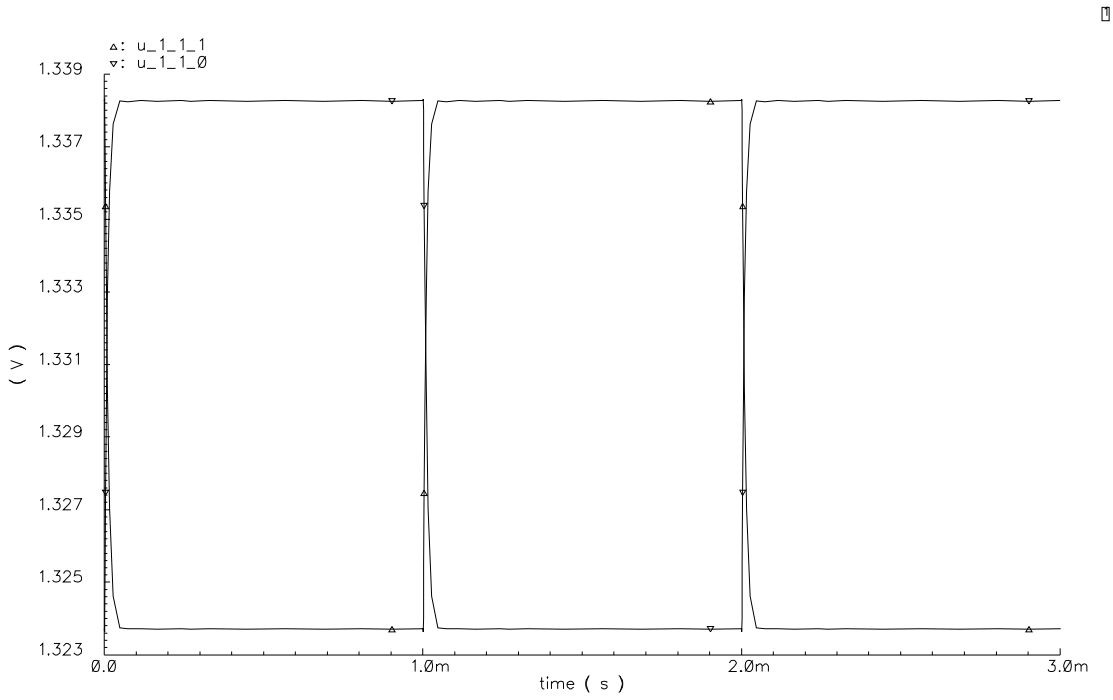

Figure 6.20. Spectre simulation result of the automatically generated schematic and layout of the product decoder.

possibly degrading performance. Actually, there are some disadvantages of our cell library. The cell library is formed by partitioning the basic building blocks into two parts, product cell and normalization cell, making the cell library require only a limited number of cells. However, there are a large number of connections between the product cell and normalization cell within one block, especially for a large building block. For example, from the trellis of the function node t of the product decoder shown in Figure 6.12, we know that in order to generate one of the outputs of function node t, one building block needs 16 inputs and 8 outputs in addition to the voltage sources. However, using our cell library, the building block is divided into a product-8-8 cell and a dnorm-8 cell. The product-8-8 cell needs 16 inputs and generates 64 outputs (all the combinations of the X input and Y input) in addition to the voltage sources. Thus there are a large number of pins needed for the product-8-8 cell, requiring huge number of connections between the cells and making automatic layout generation difficult. A possible solution is to automatically generate the layout hierarchically such as generating the layout of the basic building blocks needed for a decoder first based on the cell library and then using the layout of these building blocks to construct the layout of the decoder. However, there is not yet any commercial tool support for this.

CHAPTER 7

CONCLUSIONS

In this chapter, some conclusions are provided. Also, some future work are discussed.

7.1 Summary

Analog error control decoders are becoming more attractive because of their performance compared with their digital counterparts. However, simulation, synthesis, and circuit performance analysis of analog error control decoders have not been systematically investigated. This dissertation provides a complete design methodology for analog implementation of error control decoders including high level simulation, automatic simulation, automatic synthesis, and circuit level modeling and implementation.

For error control decoders, a large amount simulation time is required to get a bit error rate curve. Spice is an accurate simulation tool for analog circuit. However, it is too time consuming. The high-level VHDL simulation method presented in this thesis provides a good balance between accuracy and efficiency. Also, using high-level VHDL simulation, we find that using a reset circuit to reset the initial conditions for decoders with cycles can improve the performance of these decoders and this phenomenon is explained at the factor graph view of the decoder.

An automatic simulation method is provided in this dissertation. The user only needs to write the factor graph description of the decoder and a simple description of the simulation environment. Then, the tool can generate the needed VHDL simulation file. This greatly facilitates the simulation process, especially for large decoders.

Also, the dissertation provides a cell library and an automatic synthesis method. From the simple factor graph description of a decoder, its schematic and layout can be automatically generated by using the tool provided by this thesis and other commercial tools. The cell library and the automatic synthesis tool greatly speed up the design process of analog decoders.

Also, this thesis presents a number of circuit level design considerations such as speed, area, and power, and some novel circuits are provided. Circuit level modeling issues such as mismatch, internal noise, channel length modulation, quadratic behavior when working at moderate and strong inversion regions are analyzed in Chapter 5. The analysis shows that the performance degradation due to mismatch is not as serious as what people usually thought and a simple equation is given for the performance degradation due to mismatch. The analysis of the nonideal effects shows that performance of the analog error control decoder is degraded only slightly, making analog error control decoders more attractive considering its substantial potential for power savings.

7.2 Future Work

High level simulation, automatic simulation and synthesis, and circuit modeling and implementation are important issues for analog error control decoders. Although the work presented in this dissertation has presented some research results in these fields, there is much work that needs to be done to make analog decoders practical. Also, there are many interesting topics that need to be researched for analog circuit applications in the coding field. This section describes the areas that we believe to be important research problems that should be addressed.

7.2.1 Automatic High-Level Simulation

Even if high level simulation is used, the simulation of large analog decoders takes much time, especially at high SNR when the bit error rate is quite low. Importance sampling is a very efficient simulation method for high SNR applications. However, the method needs to use the error centers of the code. The automatic generation of the error centers of a code needs to be investigated to make automatic high-level simulation using importance sampling method possible.

7.2.2 Automatic Synthesis

Using the presented cell library, a method is needed to determine whether the normalization cell must provide two sets of outputs, one for the X input of product cells and one for the Y input of product cells, or only one set of them. However, this problem can only be settled when the instances of product cells have decided which input should be accepted as X input and which input should be accepted as Y input and the problem is made even more complex for hierarchical design. A possible solution is let the product cell

make this decision based on speed and area consideration and also let the normalization cell provide both X and Y outputs. Then, the wire connections of the instances of product cells and normalization cells should be extracted. If only one set of outputs of an instance of a normalization cell is used, then this instance is substituted by a normalization cell that only provides one set of outputs and redundant circuits are removed.

Also, the automatically generated circuit has longer wire connections than hand design. An automatically generated circuit needs to be fabricated and tested to do performance comparison with the corresponding hand design to see how much performance loss may be induced by the automatically generated circuit.

7.2.3 Circuit Considerations

As discussed in Chapter 5, mismatch is not a serious problem. The performance degradation due to mismatch is not large. Meanwhile, the performance degradation due to the quadratic behavior when the circuit works in strong inversion region is larger than the mismatch effect. Scaling the transistors to smaller size pushes the transistors to work more toward the subthreshold region. Thus, if the transistor size is scaled to a smaller size, the effect due to mismatch increases while the quadratic behavior decreases. Because the performance degradation due to 10 percent mismatch is still very small, using smaller transistor size, a lower power supply requirement may be possible. These problems need to be researched.

7.2.4 Space Time Coding

In this dissertation, AWGN channel is used to model the noise channel. In reality, especially in the wireless communications in which the the presented analog error control decoders is most likely to be used due to its low power, this is not true. Instead, the channel may be a constantly changing channel and space time coding theory [59], [58], [11] is used for these applications. In order to make an analog error control decoder practical, the analog implementation of space time decoding needs to be researched.

APPENDIX A

FACTOR GRAPH DESCRIPTION LANGUAGE AND SOME FACTOR GRAPH EXAMPLES

For both automatic simulation and synthesis of error-control decoders, the factor graph description of the decoder is needed as the input. This appendix describes the factor graph language using the extended Hamming (8,4,4) decoder as an example.

A.1 Compatible Language with `dot`

Of course, the factor graph is a graph. The initial concern in defining the language is choosing a graph description language that describes the nodes, edges, and the connections. However, for a large factor graph, such kind of description is too time consuming and error prone. We would like the factor graph description to be simple. Meanwhile, it can be helpful for the designer to see the graph of the factor graph and even do changes on the graph. `dot` is a powerful graph style description and drawing tool developed by Bell Laboratories. It is chosen to draw the factor graph and used to do interactive changes on the graph. The idea of showing the graph and do interactive graph editing is shown in Figure A.1 though the idea is still not implemented. In order to make the compiler easy, the language is chosen to be compatible with the `dot` style. As a result, each module always has the form shown below. Also, for each statement except the instance description statement, it always begins with special symbols "{ and ends with special symbols }" so that it is compatible with the `dot` language and also can be recognized by the compiler that it is not a graph style description. Also, every statement should be written in one line so that when the statement is too long, the symbol \ needs to be used.

```
graph module_name {
  statement;
  statement;
  ...
  statement;
  }
```

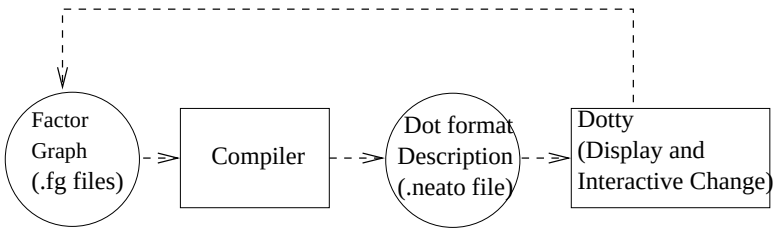

Figure A.1. Make the language compatible with `dot`.

A.2 Definition of Decoder

The decoder is described by its factor graph description. Hierarchical description is used to describe the factor graph and this is very helpful when describing a large factor graph. For the hierarchy description, the leaf node describes the trellis sections while the non-leaf nodes describe how these trellis sections are used to construct the whole factor graph. In order to distinguish these two kinds of modules, the trellis section module name always begins with the characters "function_". In the following subsections, we describe how the trellis section and the whole factor graph are provided.

A.2.1 Definition of a Trellis Section

In probability propagation networks, all the messages passed between nodes are real number probabilities. What we need to describe is how many real number probabilities are passed along an edge and we need to distinguish these probabilities. This can be accomplished by using a type statement. The following are some examples of type statements. The "bin" type means that there are two probabilities represented by the name "0" and "1" respectively.

```
‘‘{type=  bin(0,1) }’’;
‘‘{type=  quat(00,01,10,11) }’’;
‘‘{type=  state4(0:3) }’’;
```

Also, we need to describe the interface of the trellis section so that it can be used by the high level module in the hierarchy. This is done by describing the interface as port and giving the ports a name. Then the trellis connection can be described as a valid configuration space just as we have described in Chapter 3. The description of one trellis section of the extended Hamming (8,4,4) decoder is shown below. Also, notice that for the port description, the direction "in","out", and "inout" can be specified like "cs: in bin" while the default direction is "inout".

```
graph function_fa{
‘‘{type=  bin(0,1) }’’;
‘‘{type=  quat(00,01,10,11) }’’;
‘‘{type=  state4(0:3) }’’;
‘‘{function =  fa{port=cs:bin,u:bin,x:quat,ns:state4;\
relation=(0,0,00,0;0,1,11,1;1,0,01,2;1,1,10,3);} }’’;
}
```

A.2.2 Hierarchy Description

For hierarchy descriptions, low level modules are required to construct a large graph. A low level module needs to be described before its usage so that the correct usage can be checked. If the low level module description is in the same file of its usage, the description should be before the usage. If the low level module is described in another file, an include statement is needed to tell the compiler where the low level module is described. An example is shown below in which “fa.fg” is the file name. Also, notice that the type statement described in the included file is inherited. As a result, a good method of describing the type statement is describing the type statement in one file and including the file when the type statement is needed.

```
‘‘{include=fa.fg}’’;
```

Also, each module should have ports as its interface, the port interface for a trellis section is already described in the function description statement. In the other modules, the ports are described clearly using the format “port=port_name:dir type,port_name:dir type,...”. A port description example is shown below.

```
‘‘{port=x :  quat,u :  bin }’’;
```

Meanwhile, to connect the low level modules, a signal is needed, a signal description is similar to the port description except no direction description is needed. It has the format “signal=signal_name: type,signal_name:type,...”. An example is shown below.

```
‘‘{signal=s0 :  bin,s1 :  state4,s2 :  bin,s3 :  state4}’’;
```

Also, for the control flow and the encoder description, a variable is needed and the definition of a variable has a similar description format with the signal description format except that in the description, “signal” is substituted by “variable.”

For large factor graphs, sometimes it is needed to describe a bunch of ports, signals, or variables of the same type. As a result, an array description is needed. As a result, ports, signals, and variables that we introduce later can be described by using arrays and the array is defined using the format “name[int][int]...” while “name[int]” means a one dimensional array with low subscript 1 and high subscript “int”. An example of using an array is shown below.

```
‘‘{port=x[4] :  quat,u[4] :  bin }’’;
```

In order to describe the hierarchy construction, a function call statement is needed. The format for a function call statement is “instance_name [function=$\ddot{\{}$module_name(port_interface)$\ddot{\}}$]”. An example is shown below.

```
f0 [function=‘‘{ fa(port=s0,u[1],x[1],s1) }’’];
```

A hierarchy construction example is shown below.

```
graph CodeGraph {
‘‘{type=  bin(0,1) }’’;
‘‘{type=  quat(00,01,10,11) }’’;
‘‘{type=  state4(0,1,2,3) }’’;
‘‘{port=x[4] :  quat,u[4] :  bin }’’;
‘‘{include=fa.fg}’’;
‘‘{include=fb.fg}’’;
‘‘{signal=s0 :  bin,s1 :  state4,s2 :  bin,s3 :  state4}’’;
f0 [function=‘‘{ fa(port=s0,u[1],x[1],s1) }’’];
f1 [function=‘‘{ fb(port=s1,u[2],x[2],s2) }’’];
f2 [function=‘‘{ fa(port=s2,u[3],x[3],s3) }’’];
f3 [function=‘‘{ fb(port=s3,u[4],x[4],s0) }’’];
}
```

A.2.3 Operations

For the description of control flow and the description of the encoder as the simulation environment. Both algorithm operations and logical operations are needed. As a result, the following algorithm operations “+”, “-”, “*”, “/”, “**”, “%” are defined. “+”, “-”, “*”, “/” have the same meaning as any language. “**” means power. “%” means mod. Also, the following logical operations “<”, “=”, “>”, “<=”, “>=”, “not”, “and”, “or” are defined and they have the same meaning as any language.

A.2.4 Control Flow

For large decoders, there might be some duplicate structures and also some conditional structures. In order to describe such kinds of structures, loop control statements and conditional control statements should be provided. For the loop control, the for statement is provided. The for statement has the following format in which the iterative variable always begins with the initial value and increases 1 in each iteration until it reaches the stop value.

```
‘‘{for iterative_variable=initial_value:stop_value}’’;
statements;
‘‘{end for}’’;
```

The conditional control statement is provided by the if statement that has the following format in which the condition is generated by the logical operations.

```
‘‘{if condition}’’;
statements;
‘‘{end if}’’;
```

A.3 Description of the Environment Encoder

Just as described in Chapter 3, the enviroment encoder is needed to do the simulation. As a result, the encoder also needs to be described. Because the encoder may not be able to be described as a factor graph, the encoder needs to be described differently. Thus, using the same description language, we would like to use a special module name "function_encoder" for the encoder so that the encoder can be realized by the compiler. Also, many linear block encoders are described by the generator matrix. In order to provide the generator matrix, a special module name "function_gen" is used. Finally, the decoder and the encoder should be connected together and a module with special name "top" should be provided.

A.4 Special Functions

In order to provide the environment, some special functions are needed just like the C library functions. These special functions are described in the following. $exp(parameter)$, $sqrt(parameter)$ are predefined VHDL functions and the compiler can use them directly. $rand()$, $rand(int)$, $rand(int1, int2)$ are provided by predefined library. $rand()$ returns a random distributed 0 or 1. $rand(int)$ returns a length int vector of random distributed 0 or 1. $rand(int1, int2)$ returns a $int1 * int2$ matrix of random distributed 0 or 1. $randn(real)$, $randn(real, int)$, $randn(real, int1, int2)$ are provided by predefined library. $randn(real)$ returns a gaussian noise with variance $real$. $randn(real, int)$ returns a length int vector of gaussian noise with variance $real$. $randn(real, int1, int2)$ returns a $int1 * int2$ matrix of gaussian noise with variance $real$. $convert(array)$ is used to convert the result of the decoder (usually a binary array) into a real array so that we can compare it with the source information that the encoder uses. $compare(source, result)$ is used to compare the $source$ with the $result$ so that we know the error rate.

Also, some special functions can be used on arrays with real elements: $row(array\_name, int)$ means the int row of $array\_name$ is used. $column(array\_name, int)$ means the int column of $array\_name$ is used. $row\_equal(array\_name, vector, int)$ means we build a new array that has all the elements equal the elements of $array\_name$ except the int row is substituted by $vector$. $column\_equal(array\_name, vector, int)$ means we build a new

array that has all the elements equal the elements of *array_name* except the *int* column is substituted by a *vector*. *row_swap*(*array_name*, *int*1, *int*2) means we build a new array that has all the elements equal the elements of *array_name* except the *int*1 row and *int*2 row are exchanged. *column_swap*(*array_name*, *int*1, *int*2) means we build a new array that has all the elements equal the elements of *array_name* except the *int*1 column and *int*2 column are exchanged. *row_move*(*array_name*, *int*1, *int*2) means we build a new array that has all the elements equal the elements of *array_name* and then the element of *int*1 row is moved to *int*2 row. *column_move*(*array_name*, *int*1, *int*2) means we build a new array that has all the elements equal the elements of *array_name* and then the element of *int*1 column is moved to *int*2 column.

A.5 Description of the Extended Hamming (8,4) Decoder

```
graph function_fa{
  ‘‘{type=  bin(0,1) }’’;
  ‘‘{type=  quat(00,01,10,11) }’’;
  ‘‘{type=  state4(0:3) }’’;
  ‘‘{function =  fa{port=cs:bin,u:  out bin,x:  in quat,ns:state4;\
  relation=(0,0,00,0;0,1,11,1;1,0,01,2;1,1,10,3);} }’’;
  }
  graph function_fb{
  ‘‘{type=  bin(0,1) }’’;
  ‘‘{type=  quat(00,01,10,11) }’’;
  ‘‘{type=  state4(0:3) }’’;
  ‘‘{function =  fb{port=cs:state4,u:  out bin,x:  in quat,ns:bin;\
  relation=(0,0,00,0;0,1,11,1;1,0,11,0;1,1,00,1; \
  2,0,10,0;2,1,01,1;3,0,01,0;3,1,10,1);}}’’;
  }
  graph CodeGraph {
  ‘‘{type=  bin(0,1) }’’;
  ‘‘{type=  quat(00,01,10,11) }’’;
  ‘‘{type=  state4(0,1,2,3) }’’;
  ‘‘{port=x[4] :  in quat,u[4] :  out bin }’’;
  ‘‘{include=fa.fg}’’;
  ‘‘{include=fb.fg}’’;
  ‘‘{signal=s0 :  bin,s1 :  state4,s2 :  bin,s3 :  state4}’’;
  f0 [function=‘‘{ fa(port=s0,u[1],x[1],s1) }’’];
  f1 [function=‘‘{ fb(port=s1,u[2],x[2],s2) }’’];
  f2 [function=‘‘{ fa(port=s2,u[3],x[3],s3) }’’];
  f3 [function=‘‘{ fb(port=s3,u[4],x[4],s0) }’’];
  }
  graph function_pdf{
  ‘‘{type=  quat(00,01,10,11) }’’;
  ‘‘{function =  pdf{parameter=n0 :  real; \
  port=y[2]:  in real,x:  out quat;\
  variable=p0:real,p1:real; \
  p0 =1.0 / (1.0 + exp(-4.0 * y[1] / n0)); \
  p1 =1.0 / (1.0 + exp(-4.0 * y[2] / n0)); \
  x[00] =(1.0 - p0) * ( 1.0 - p1); \
```

```
x[01] =(1.0 - p0) * p1; \
x[10] =p0 * (1.0-p1); \
x[11] =p0 * p1; } }'';
}
graph decoder {
``{type=  bin(0,1) }''
``{type=  quat(00,01,10,11) }''
``{parameter=n0 :  real}'';
``{port=y[8] :  in real,u[4] :  out bin }'';
``{include=CodeGraph.fg}'';
``{include=pdf.fg}'';
``{signal=x[4] :  quat}'';
``{ for i=1:4 }'';
pdfi[function=''{pdf(parameter=n0;port=y[2*i-1:2*i],x[i]) }''];
``{ end for }'';
TheCodeGraph[function=''{CodeGraph(port=x[1:4],u[1:4])}''];
}
graph function_gen{
``{function= gen{port=s[4] :  in real,x[8] :  out real; \
variable=genarray[4][8] :  real; \
genarray={{1.0,1.0,1.0,1.0,0.0,0.0,0.0,0.0},{0.0,0.0,1.0,1.0,0.0,1.0,1.0,0.0},\
{0.0,0.0,0.0,0.0,1.0,1.0,1.0,1.0},{0.0,1.0,1.0,0.0,0.0,0.0,1.0,1.0}}; \
x=s * genarray % 2.0; \
} }'';
}
graph function_encoder{
``{type=  bin(0,1) }'';
``{include=gen.fg}'';
``{function = encoder{parameter=n0 :  real; \
port=u[4] :  in bin,y[8]:  out real; \
variable=s[4] :  real; \
variable=olds[4] :  real; \
olds=s; \
s=rand(4); \
y=2.0*gen(s)-1.0+randn(sqrt(n0/2.0),8); \
compare(convert(u),olds); }}'';
}
graph top{
``{type=  bin(0,1) }'';
``{type=  quat(00,01,10,11) }'';
``{signal= n0 :  real}'';
``{include=decoder.fg}'';
``{include=encoder.fg}'';
``{signal=u[4] :  bin,y[8] :  real}'';
en [ function=``{ encoder(generic=7.0,4.0,0.5,20.0;parameter=n0;port=u,y) }'' ];
de [ function=``{ decoder(parameter=n0;port=y,u) }'' ];
}
```

APPENDIX B

DESIGN FLOW FOR AUTOMATIC SYNTHESIS AND AUTOMATIC SIMULATION

Chapter 1 describes the design flow for automatic synthesis and simulation. This appendix provides a detailed description of how to use the compiler and cell library in this thesis along with `Cadence` and `Synopsys` to generate the high level simulation result and the schematic and layout of a decoder is presented. This appendix is described in a tutorial style to explain the detailed design flow shown in Figure B.1. The user can use this appendix as a tutorial for doing automatic synthesis and simulation of analog error control decoders.

B.1 Generating VHDL Files by Using the Compiler

The first step is to generate VHDL files using the compiler, analog_sim or analog_syn, provided by this thesis. The user should provide the factor graph description of the decoder. Also, the user should also provide the description of the simulation environment if he or she is doing simulation. For the factor graph description of the decoder and the description of the simulation environment, please see Appendix A.

For simulation, there should be a top level description, such as top.fg, descripting how to connect the decoder and the encoder. Also, all the description files of the decoder and the encoder should be in the same directory of the top level description file. Then, the user should type analog_sim top.fg for the compiler to generate all the VHDL simulation files. All the resulting VHDL files are generated in the same directory as the top level description file top.fg.

For synthesis, the user should notice that the simulation environment is not able to be synthesized. Only the decoder part can be synthesized. The user can synthesis the whole decoder as well as any part of the decoder. Also, the description of the analog

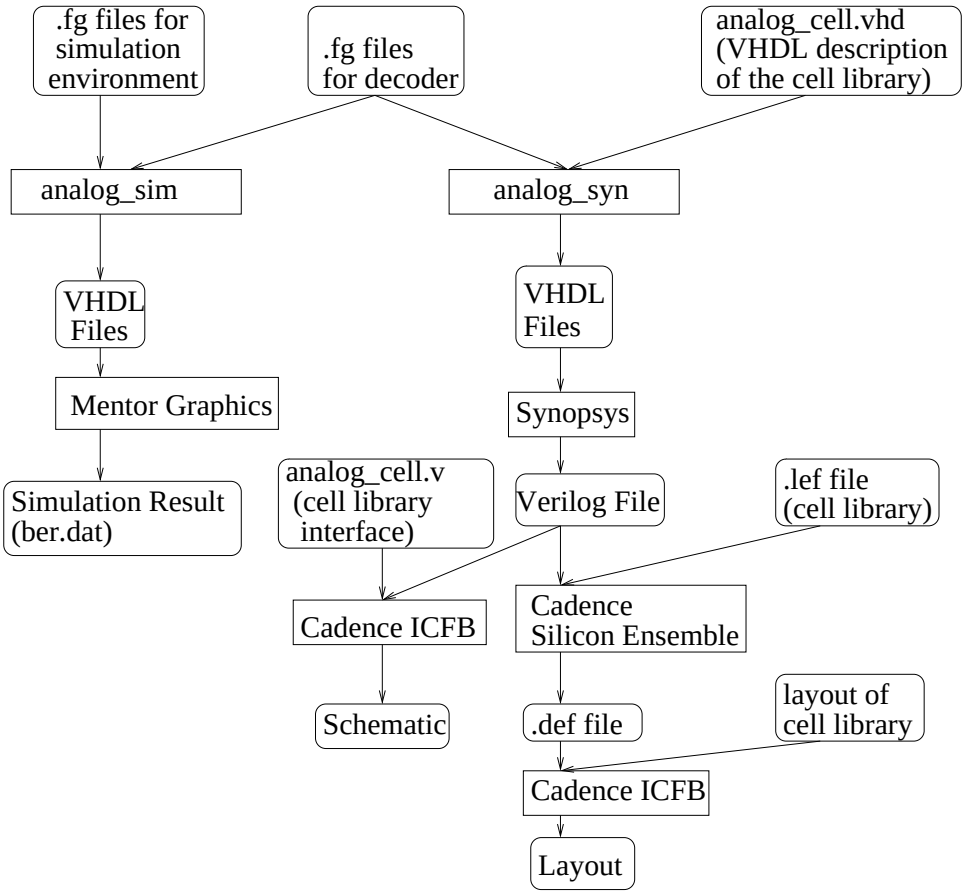

Figure B.1. Detailed design flow for automatic synthesis and simulation.

cells as component are provided in file analog_cell.vhd and this file should be in the same directory as the .fg files. The user should type analog_syn input.fg for the compiler to generate the resulting VHDL file. The resulting VHDL file is generated in the same directory as the input file.

B.2 Simulation

For simulation, besides the VHDL file generated by the compiler, the user still needs to use three VHDL files, nondeterminism.vhd and gaussian.vhd and myarray.vhd. These three files are library files, and they provide the functions that are used to do simulation. After compiling all these VHDL files using Mentor Graphics, the user can do the high level simulation for the decoder. The result is stored in the file ber.dat. The first column of ber.dat is the E_b/N_0 data, the second column is the bit error rate. The user can use Matlab to show the simulation result.

B.3 Synthesis

The synthesis flow is much more complex than the simulation flow. There are several steps as shown in Figure B.1, including generating the Verilog file, generating the schematic, using Silicon Ensemble to do place and route and use ICFB to finish the layout. Also, the user can do Spice simulation on the generated circuit. All these works are described in the following corrsponding subsections.

B.3.1 Convert VHDL Files to Verilog Files

Because `Silicon Ensemble` uses Verilog struct file as input. The first step is VHDL to Verilog conversion. This can be easily accomplished using `Synopsys`. The user can start up `design_analyzer`. Then, select the File→Read option from the menu. This brings up a dialog box to read in the design file. Choose the generated VHDL file of the decoder. Then click OK to read in the VHDL code. Notice that if there are multiplie VHDL files for the decoder, all of them should be read in. After all of the files are read, there should be a yellow box for each entity with its name in the corresponding box. Then, click on the entity that needs to be synthesized. The outline of the entity should be changed to dotted. Now, select the File→Save as option from the menu. This brings up a dialog box to save the design file. Choose the File Format as Verilog, type the name such as decoder.v, make sure to check the Save as a whole file box and then choose OK. Now, a structural Verilog description of the decoder should be generated. Another

benefit of using Synopsys is that all the low level Verilog modules (corresponding to the entities of VHDL) are all saved in one file so that the resulting Verilog file is a complete description of the decoder.

B.3.2 Generating the Schematic

Get to a point where icfb is running with the CIW. Select the File→Import→ Verilog command in the CIW. This brings up a dialog box. In the "Target Library Name" slot, put the name of the library that will be used to hold the decoder. In the "Reference Library" slot, add analog_cell to the front of the list. The user can get rid of the sample library if he or she want, or leave it in, it doesn't matter, but make sure to leave the basic library in the list. In the "Verilog Files to Import" slot put the name of your structural Verilog file such as decoder.v that came from Synopsys. Also, in the "-v Options" slot put analog_cell.v, which is a file that has interface descriptions of the analog cell library provided by this thesis. Leave other options unchanged and click OK. After several seconds, the schematic of the decoder should be generated.

B.3.3 Generating Layout

To generate the layout, Silicon Ensemble need to be used first to generate the placed and routed circuit. Then, Virturoso is used to read in the placed and routed circuit and do some simple revise to generate the layout.

B.3.3.1 Place and Route with Silicon Ensemble

1. Before starting up Silicon Ensemble, it is better to make a directory named dbs in your work directory. Silicon Ensemble needs this directory and it is not very good at making it itself. Then, start up Silicon Ensemble by typing sedsm.

2. The first step in using Silicon Ensemble is to load the information about the analog cell library. This information is in the analog_cell.lef file provided by this thesis. Select File→Import→LEF to get the right dialog box. Select analog_cell.lef and also click the "clear existing design data" button. Then Click OK.

3. The next step is to load a Verilog file analog_cell.v provided by the thesis that has interface descriptions of all the analog cells in Verilog. Select File→Import-Verilog to get the dialog box. Fill in the box with analog_cell.v in the Verilog Source Files slot. Leave the Top Module blank. Leave other things the same, and click OK.

4. Now, load the structural Verilog file that describes the circuit that you want to place and route. Choose File→Import→Verilog, but this time put decoder.v as the file and the name of the top-level module decoder at the Top Level Module before clicking OK. By the way, if a dialog box comes up asking if it is all right to overwrite or augment the existing cds_vbin library, say OK.

5. Now, the floorplan need to be initialized for the placed and routed circuit. Select Floorplan→Initialize Floorplan form the menu that brings up a large dialog box. Choose an appropriate I/O to Core Distance such as 40.00 each. Because the cells in the cell library have many pins (The product_8_8 cell has 81 pins besides vdd! and gnd!), the Row Spacing and Row Utilization should not be high to avoid routing trouble. Generally, the Row Spacing should not be smaller than 10 microns and the Row Utilization should not be larger than 70. Depending on how large your design is, choose the appropriate value and then click OK.

6. Now, select Place→IOs from the menu. The user can select Random or give constraint.

7. Now, the user needs to plan for the power lines. Select Route→PlanPower from the menu. A box should pop up. Selcet the Add Rings selection in the box. Things should be set up with gnd! and vdd! as the Nets, and M1 as the horizontal layer and M2 as the vertical layer. Change them if they are not. Then, select a multiplier of the λ for the Core Ring Width on both M1 and M2 (For example, if the technology the user are using has $\lambda = 0.3$ micron, the user can choose 4.5 micron.) and on the Block Ring Width. Keep the Core Ring as Center for both and then click OK.

8. Now, select Place→Cells from the menu. The user can leave all the boxes unchecked and just say OK or choose some options to generate a better result.

9. Now, the user needs to add filler cells. Select Place→FillerCells→AddCells to get the dialog box. Fill in the model as FILL and the prefix as fill. Uncheck all the Placement boxes except North and then click OK.

10. Now, the user needs to connect the vdd! and gnd! lines in the standard cell rows to the bus lines around the cells. Select Route→Connect Ring. Leave all the boxes as default and click OK. This routes all the vdd! and gnd! lines in the cell rows to the power buses around the edge.

11. Now, the user can route the circuit. Select Route→WRoute to perform global and final routing in one step. Make sure the Global and Final Route is selected, and the Auto Search and Repair is also selected. Click OK. This can take a long time. After routing, make sure there is no geometry violations.

12. If there are geometry violations, check whether the route has being performed correctly. If there are no geometry violations, export the design to a DEF file by choosing File→Export→DEF and type the selected name such as decoder.def. Now, the work of Silicon Ensemble has been done and the user should have a placed and routed circuit.

B.3.3.2 Reading the Placed and Routed Layout into Virtuoso

Now, the user has the placed and routed circuit. However, you still need to import the .def file into Virtuoso to get a real layout.

1. First start up icfb again and fire up the library manager. Select File→Import→DEF. In the dialog box fill in the library name as the library that the user is reading the design into. The cell name should be chosen to have the same cell name as the schematic for further usage. The view name must be autoRouted. Click the Use Ref. Library Names box and fill in analog_cell as the ref library. Fill in the name of the user's def file such as decoder.def. Make sure the Silicon Ensemble is selected as the Target P&R engine and then click OK.

2. Now, go to the library manager and click on the cell that the user has imported. The user should see an autoRouted view. Open the autoRouted view and then save it as layout view. Close the autoRouted view and open the layout view.

3. Then choose Floorplan→Replace View menu. A dialog box should pop up. Choose Replace to as layout and check the all box and then click Apply. Close the dialog box. Choose Tools→Layout menu. This changes the tool from abstract-editing mode to layout-editing mode.

4. Now, choose Create→Pin menu to put M1PIN input pins with vdd! and gnd! names on the vdd! and gnd! rings. (The vdd! ring should be the outside ring and gnd! ring should be the inside ring).

Now, the layout is finished and the user should save it. Also, the user can do DRC check of the layout to verify there is no DRC errors. The user can also generate an extract view for further use.

B.3.4 Simulation of the Generated Circuit

Now, the user should have a schematic view and layout view of the decoder. The user can create a symbol from the schematic of the decoder and then create a test bench circuit to do Spice simulation of the decoder. For the Spice simulation of the layout. The user can do an LVS check of the layout first to verify that the layout is correct. Then, the user can build an analog_extracted view of the decoder. With this view, the user can create a config view of the test bench circuit and select analog_extracted view for the decoder symbol in the config view. Then, the user can do Spice simulation of the circuit based on its layout. Using this method, the user can even compare the Spice simulation result of the schematic and layout.

B.4 Miscellaneous Things

The previous sections have described the whole flow for automatic synthesis and simulation of error control decoders. However, the .lef file is already provided by the author. Actually, generating this .lef file is not easy. If the user would like to create the layout of the cell library themselves, there are a large number of factors to be considered. The user should take careful considerations when drawing the layout of the cell library and then generating the abstract of the cell library. Fortunately, there is a good guideline for a standard cell library:

$http://www.ece.msstate.edu/EE8273/lectures/stdcellroute/stdcellroute.pdf$

and how to generate the abstract of the cell library:

$http://www.erc.msstate.edu/mpl/education/cadence/standard_cell/downloads.html$

Also, there is also a good tutorial provided by Professor Erik Brunvand and you can look at it to get a better understanding of using Silicon Ensemble.

$http://www.cs.utah.edu/classes/cs5710/labs/synopsys.htm$

APPENDIX C

SCHEMATIC, LAYOUT, INTERFACE OF CELL LIBRARY AND VHDL EXAMPLE

In this appendix, the schematic, layout, interface of the cell library is given. Also, the automatically generated VHDL files for the Hamming (8,4) decoder is given as an example.

C.1 Schematic of Cell Library

The schematic of the current cell library is shown in Figure C.1 to Figure C.11.

C.2 Layout of Cell Library

The layout of the current cell library is shown in Figure C.12 to Figure C.18.

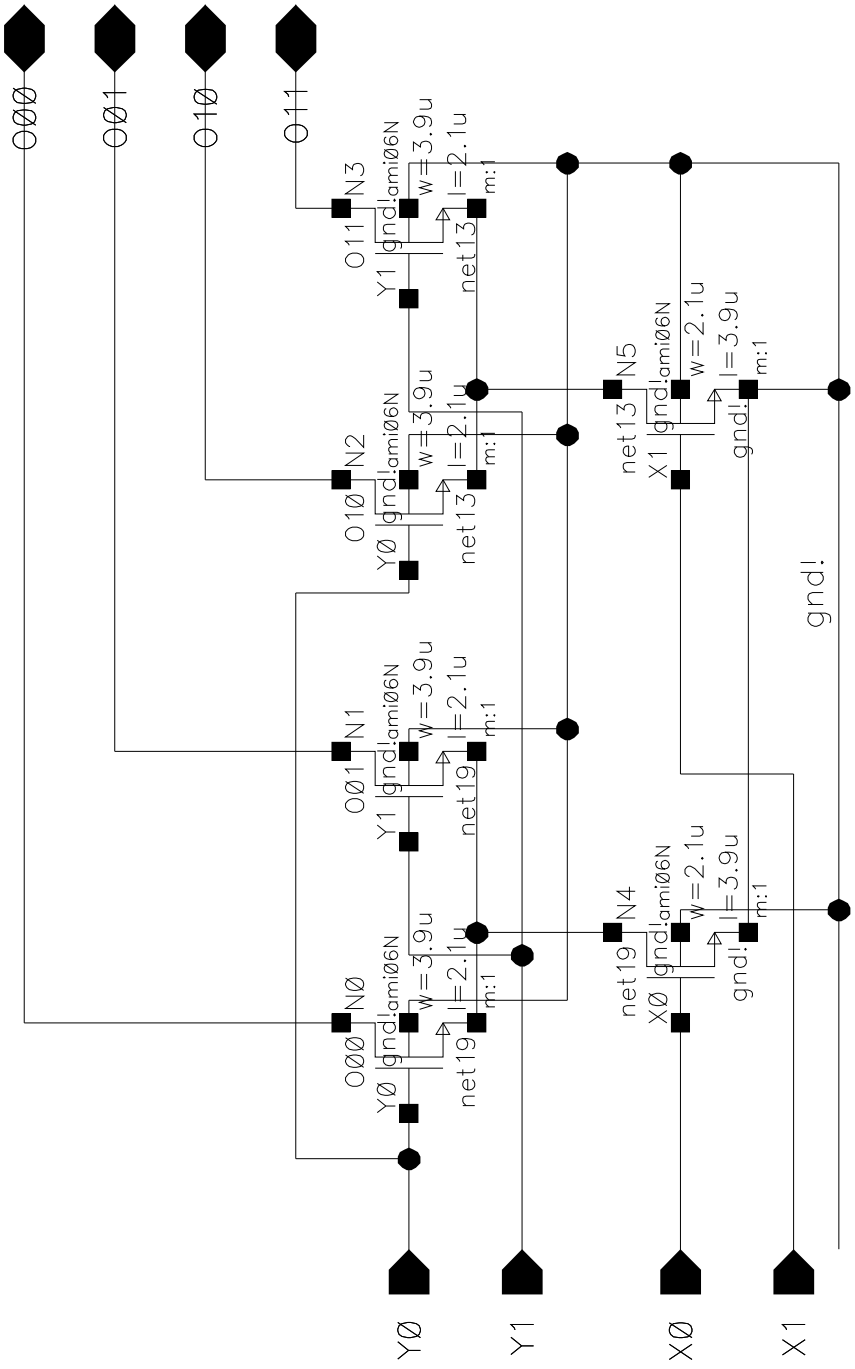

Figure C.1. Schematic of the product_2_2 cell.

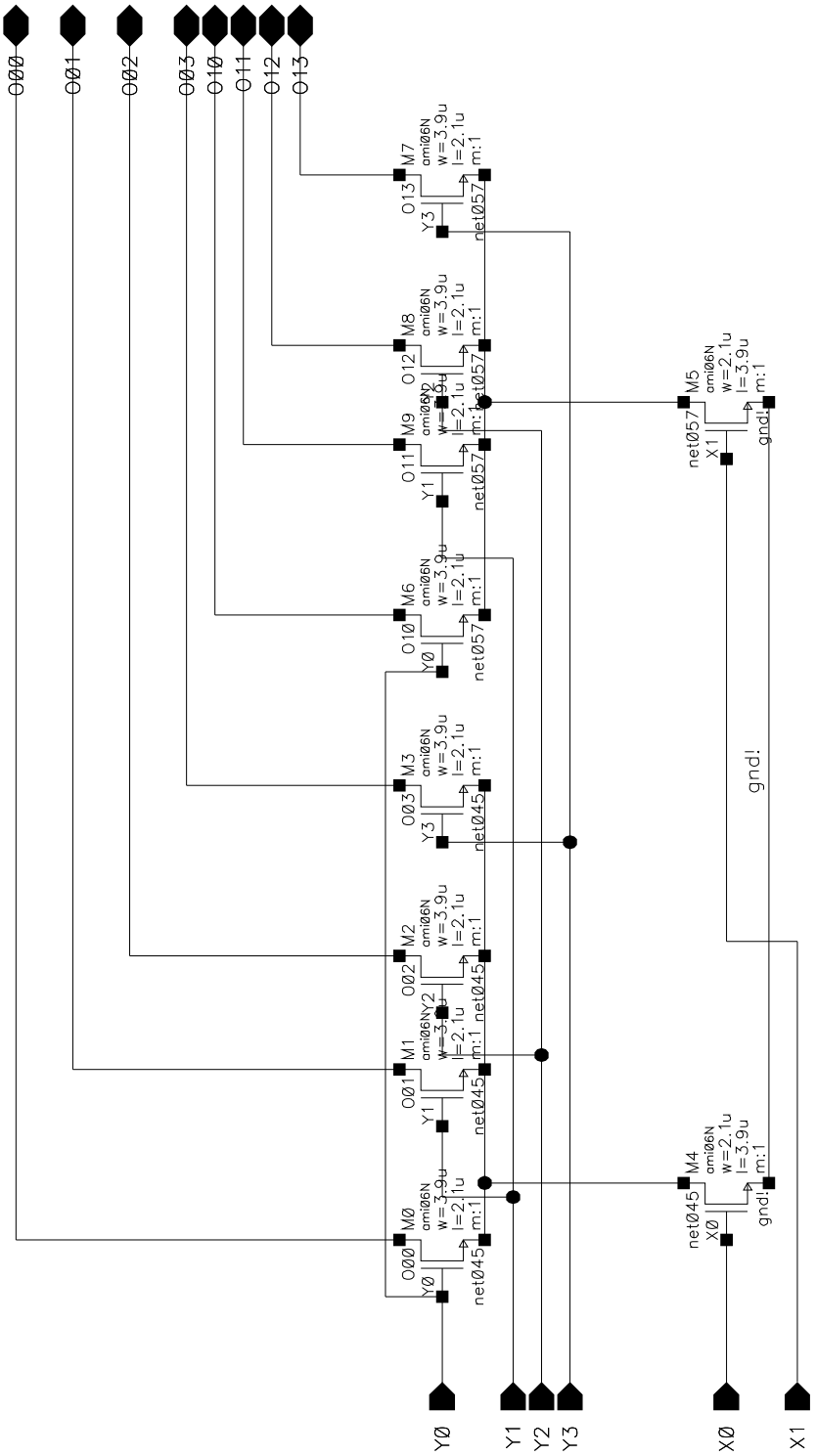

Figure C.2. Schematic of the product_2_4 cell.

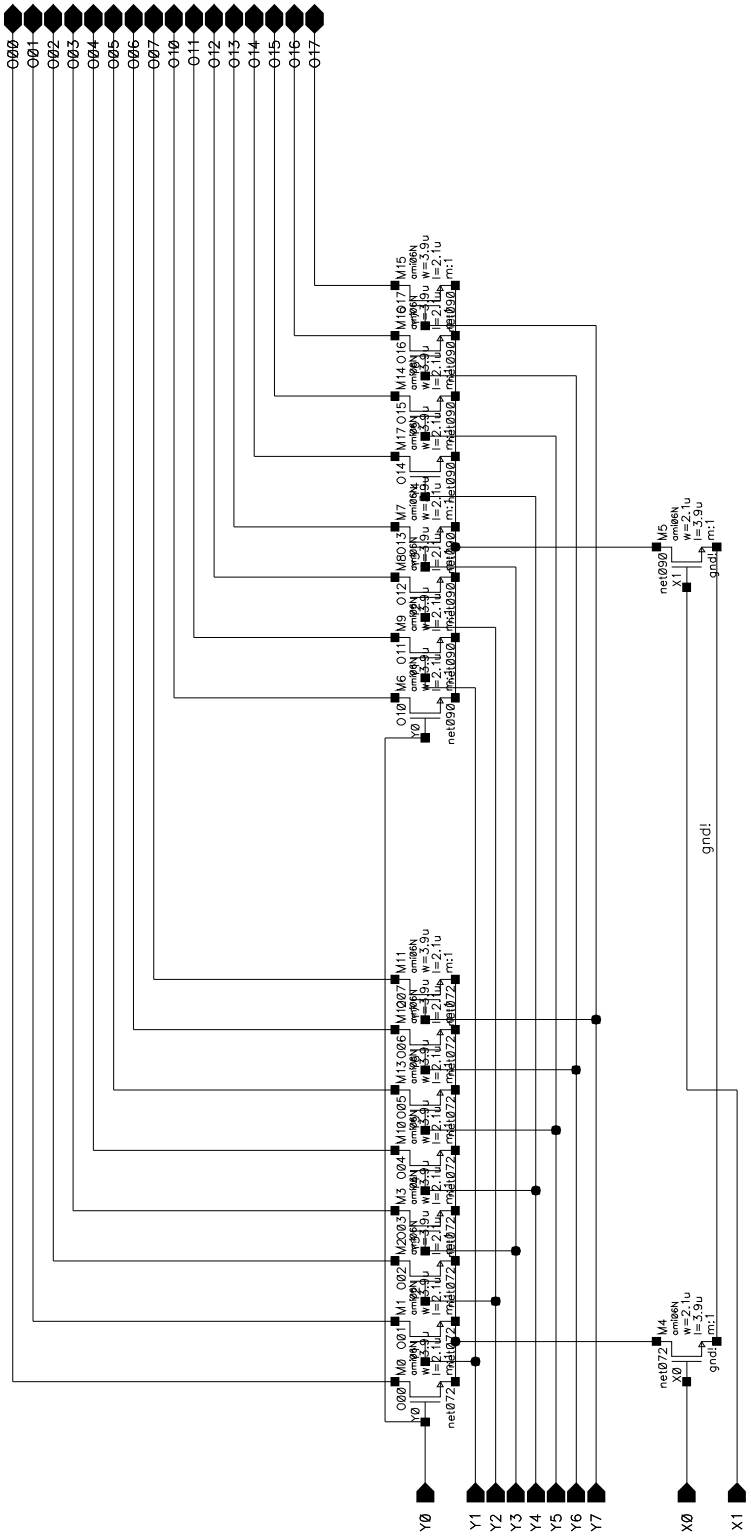

Figure C.3. Schematic of the product_2_8 cell.

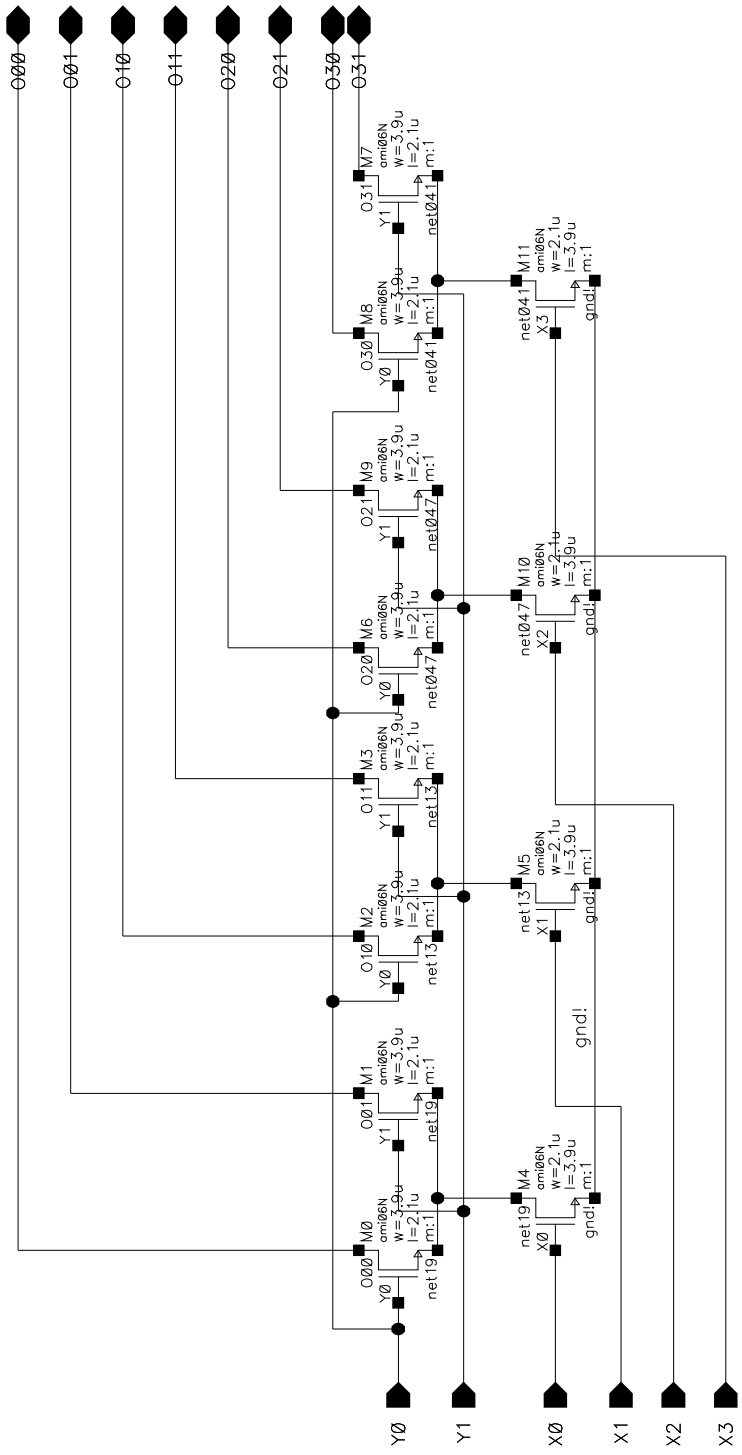

Figure C.4. Schematic of the product_4_2 cell.

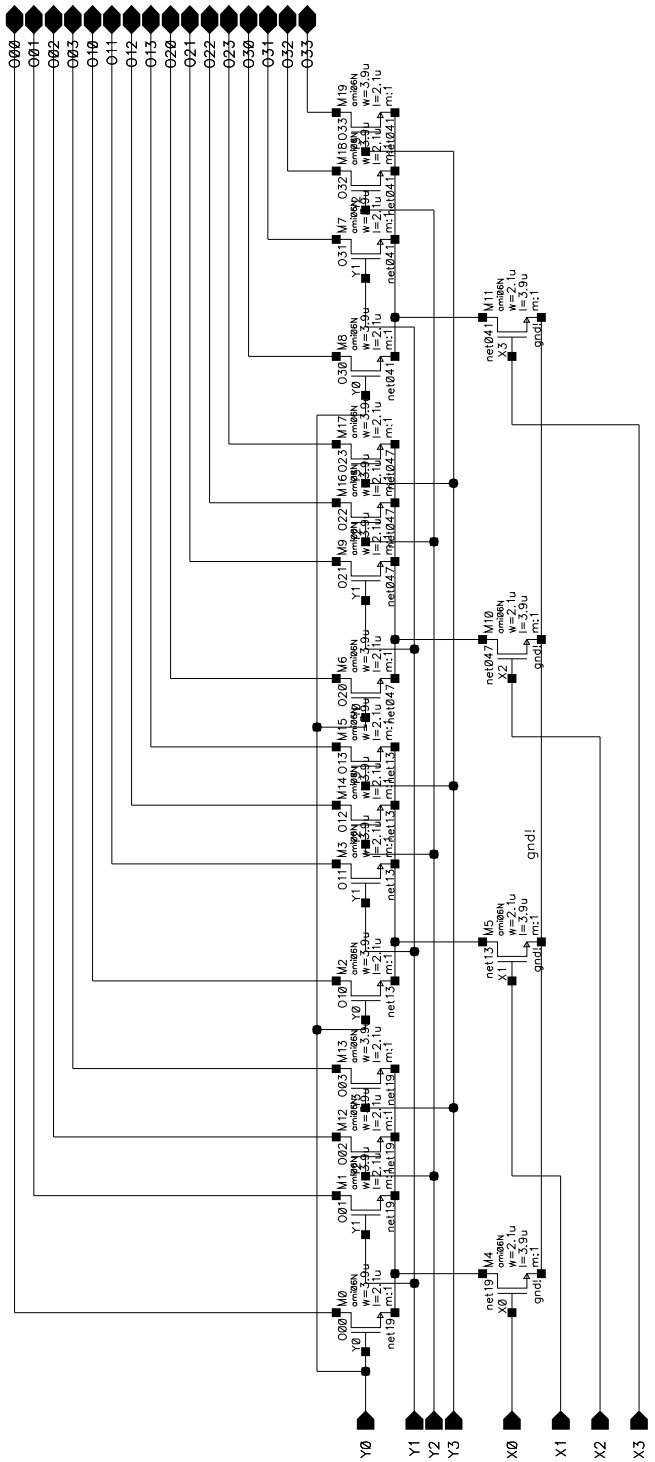

Figure C.5. Schematic of the product_4_4 cell.

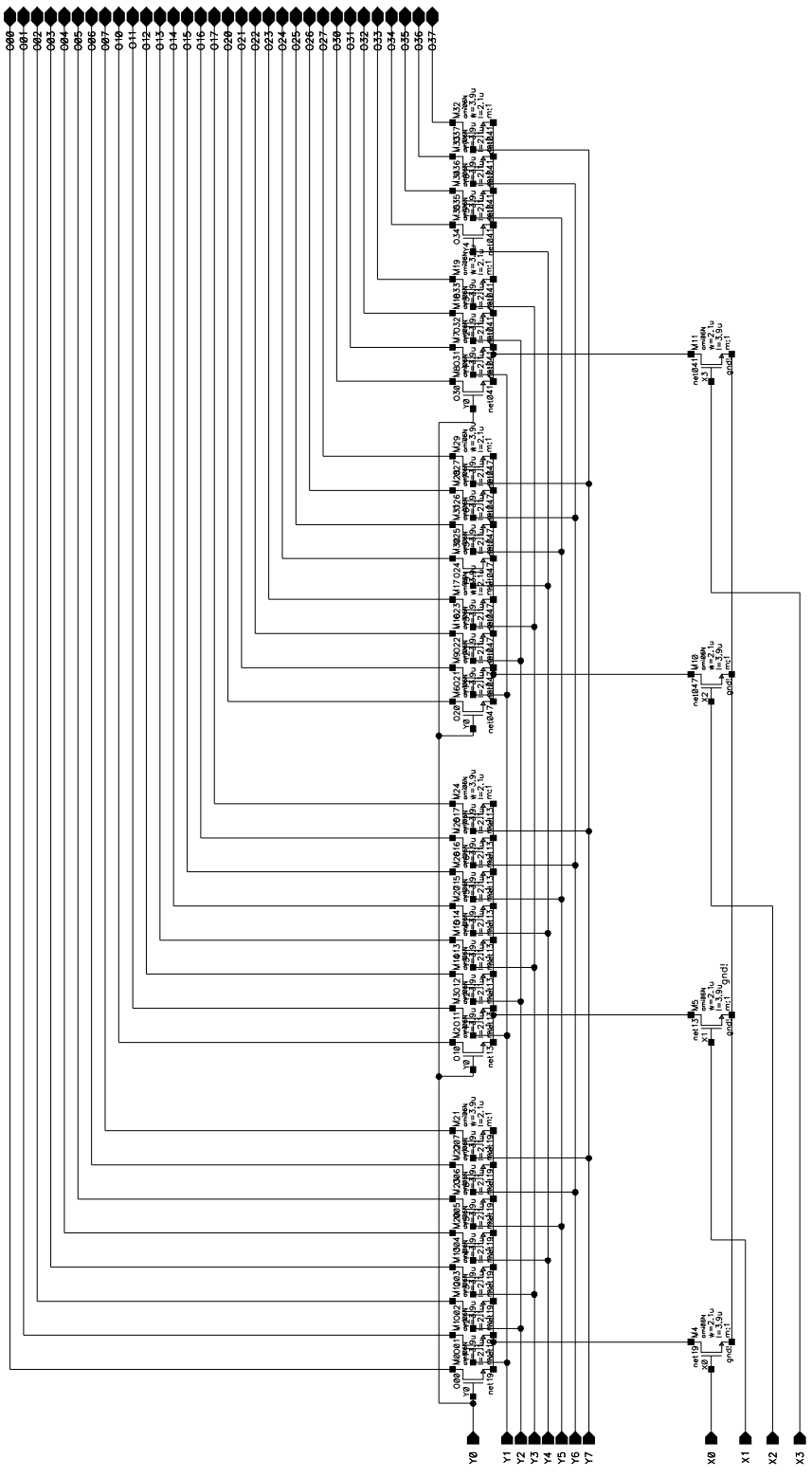

Figure C.6. Schematic of the product_4_8 cell.

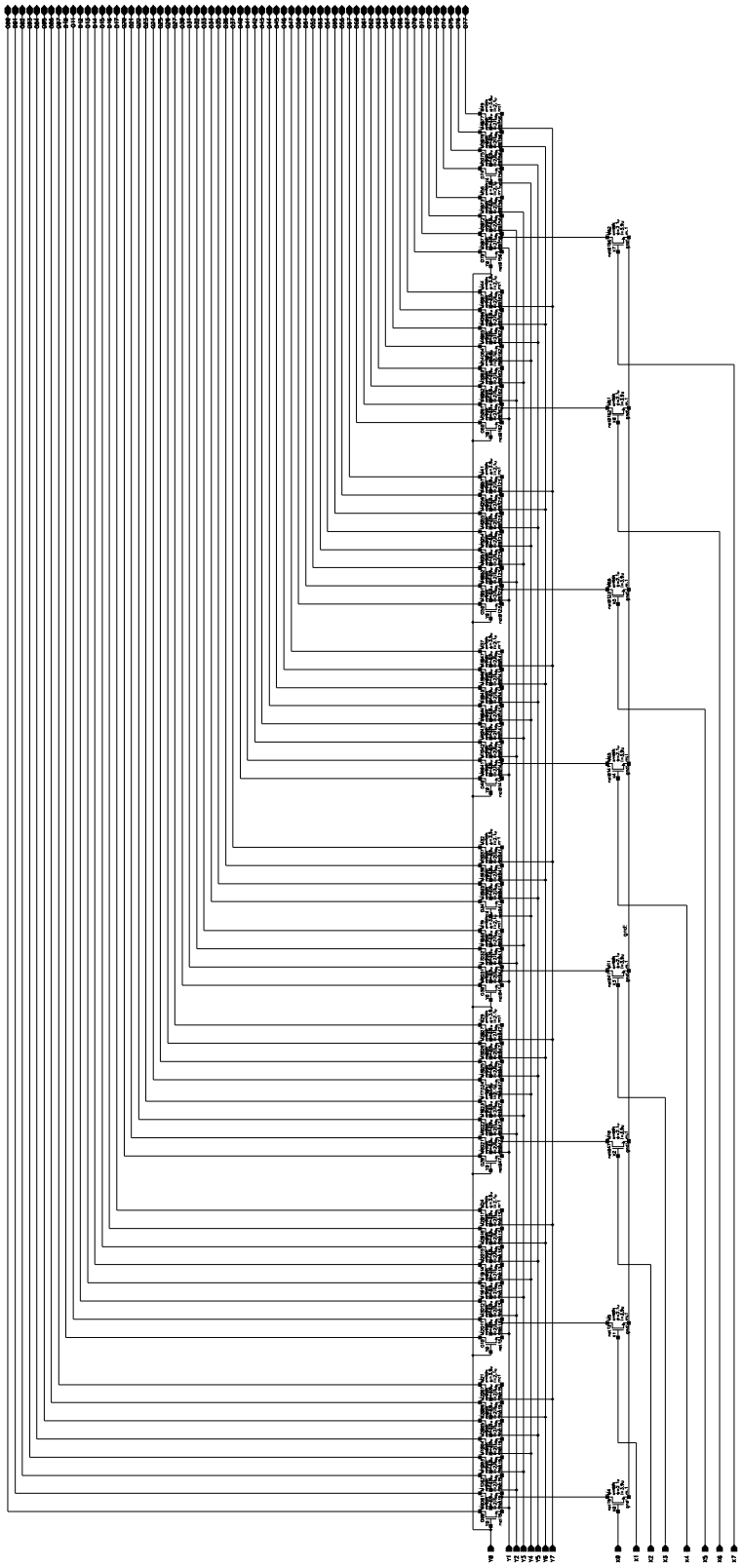

Figure C.7. Schematic of the product_8_8 cell.

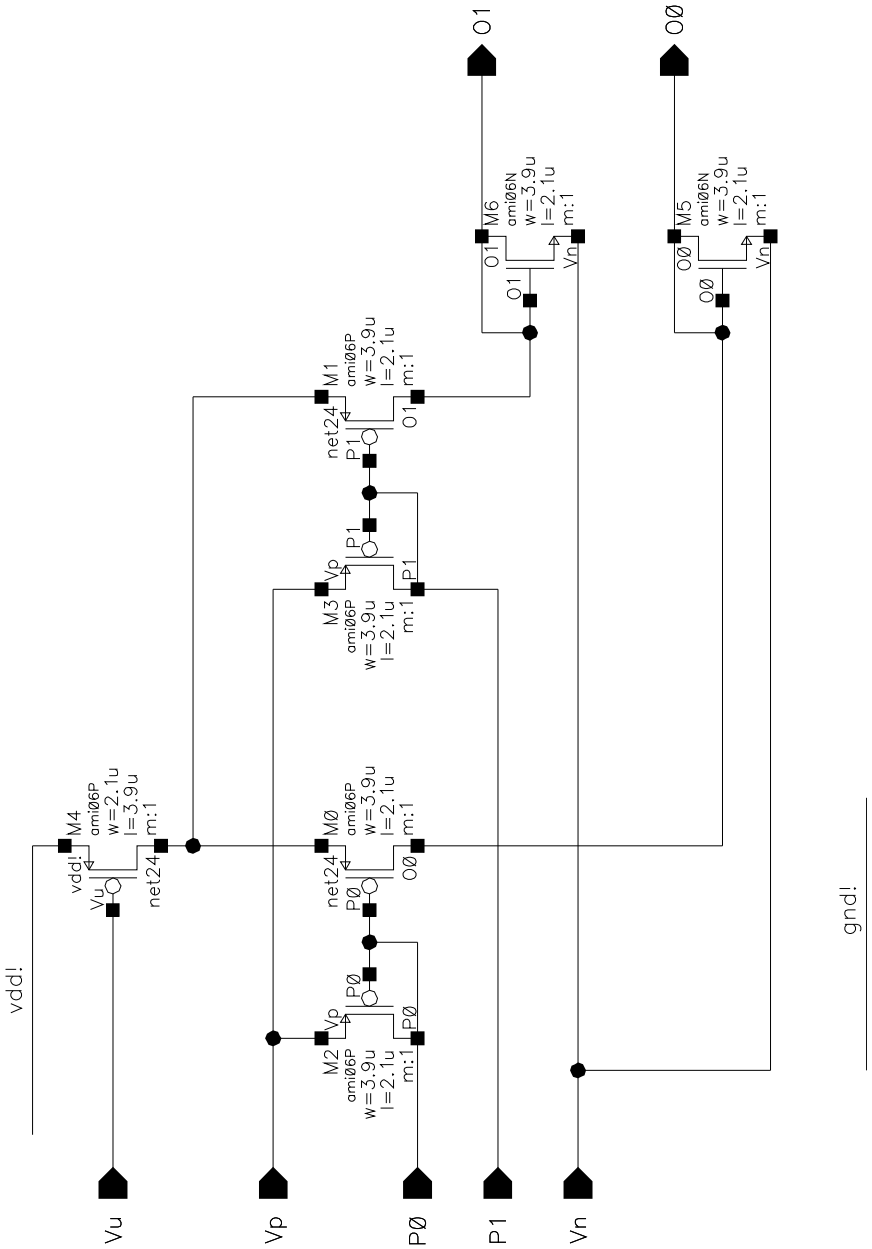

Figure C.8. Schematic of the norm2 cell.

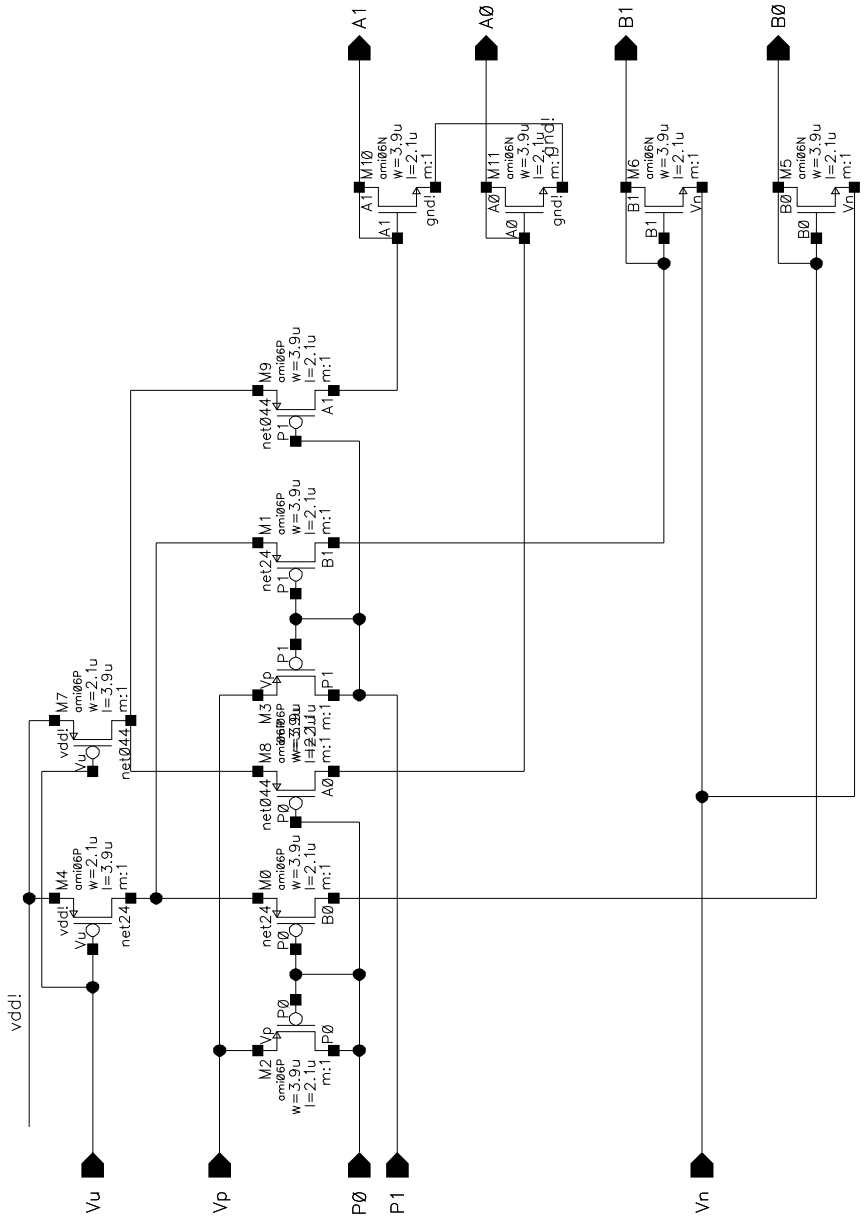

Figure C.9. Schematic of the dnorm2 cell.

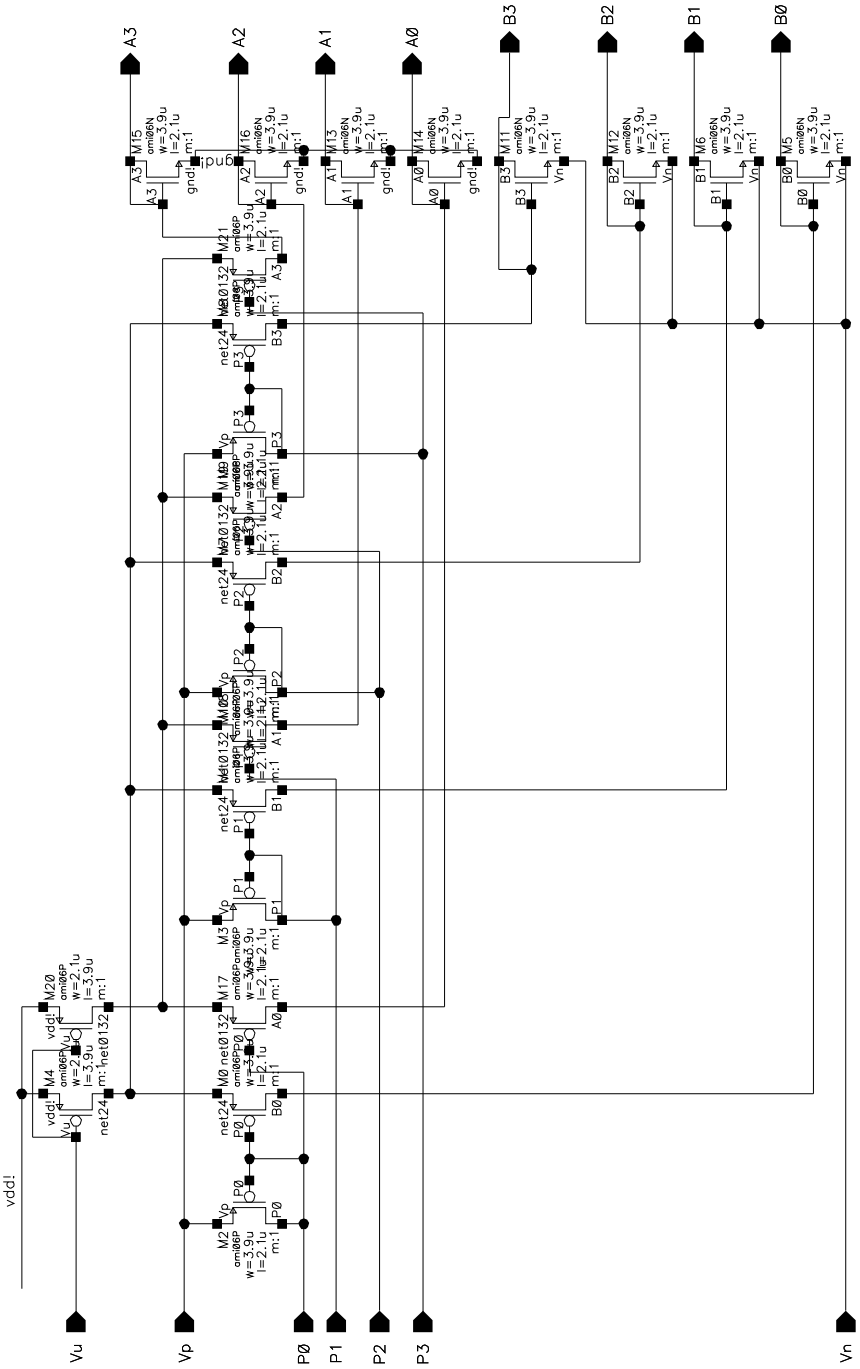

Figure C.10. Schematic of the dnorm4 cell.

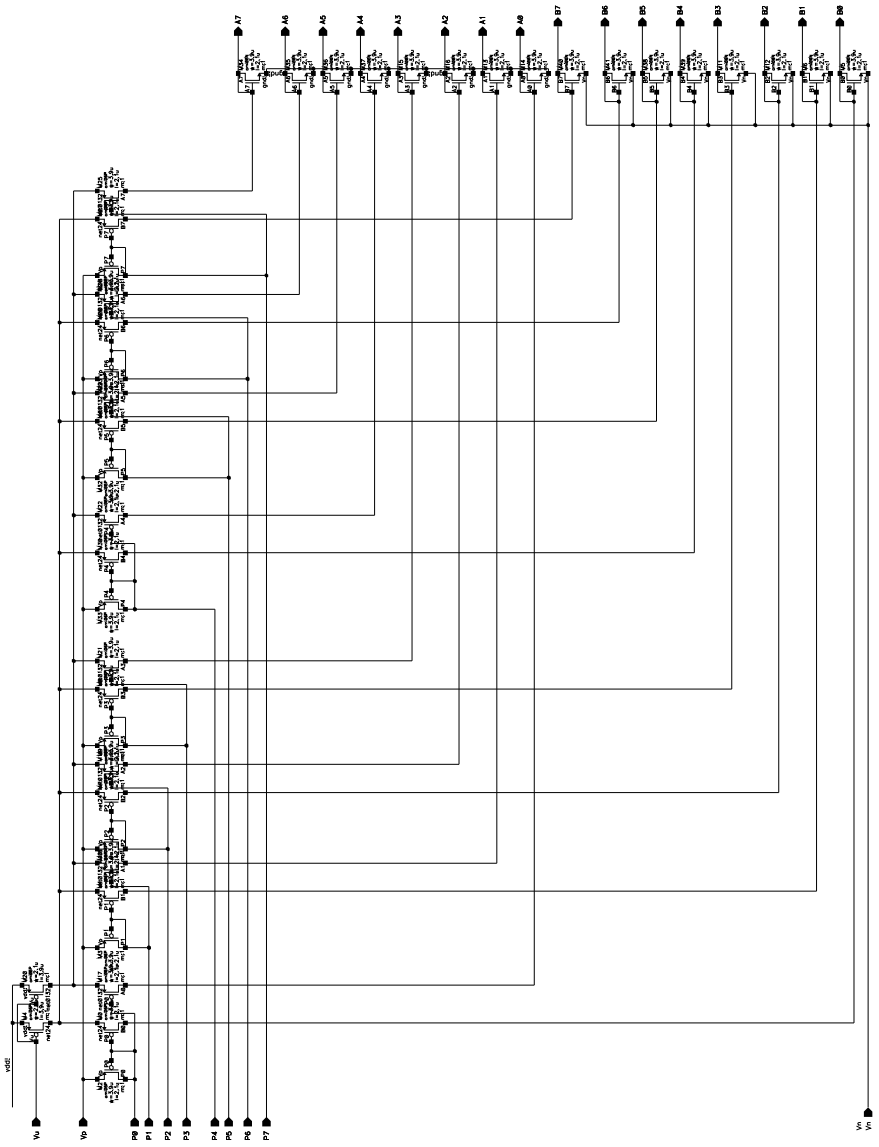

Figure C.11. Schematic of the dnorm8 cell.

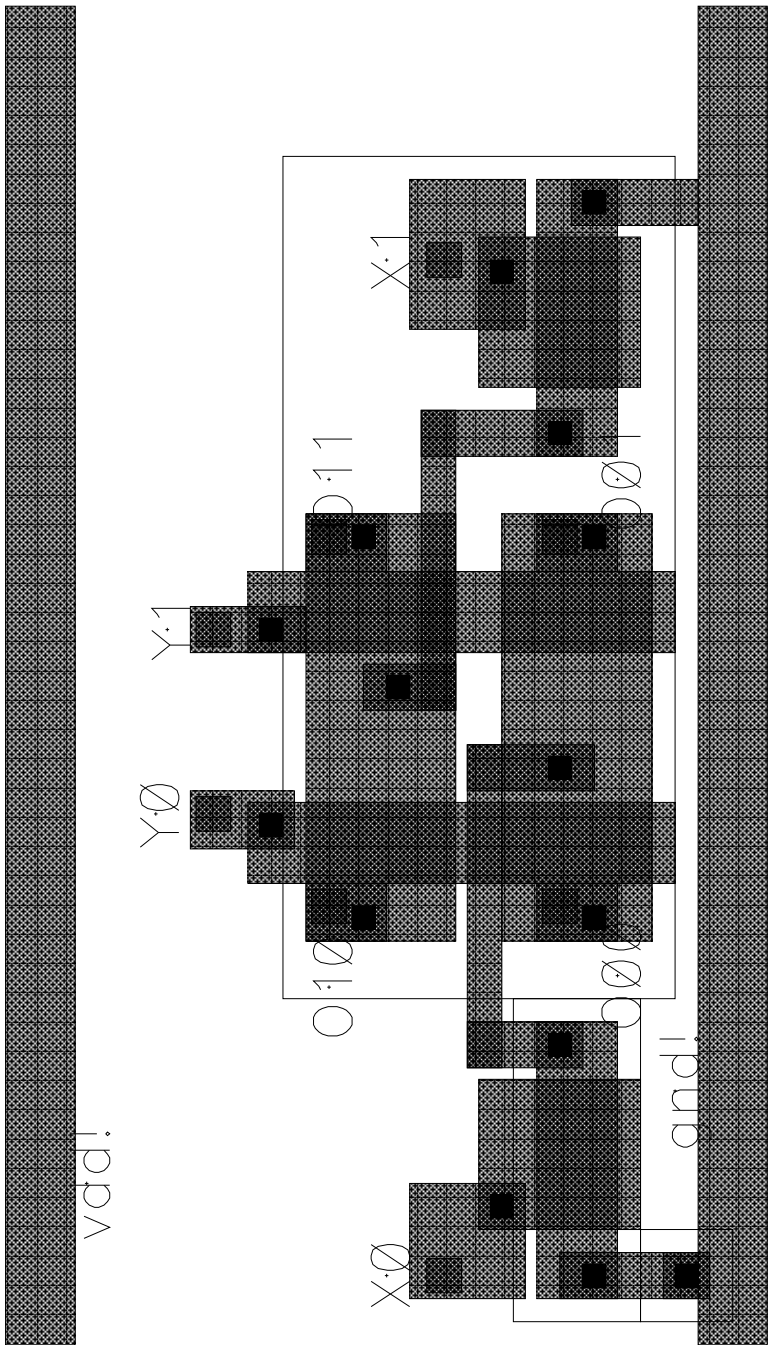

Figure C.12. Layout of the product_2_2 cell.

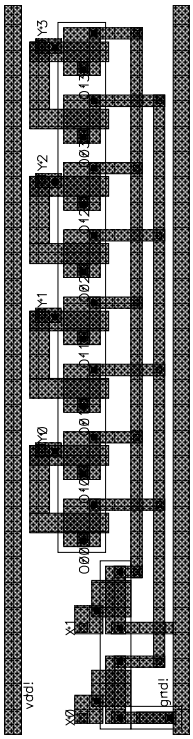

Figure C.13. Layout of the product_2_4 cell.

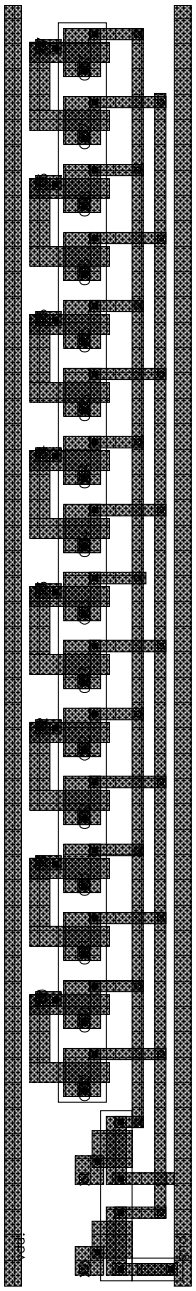

Figure C.14. Layout of the product_2_8 cell.

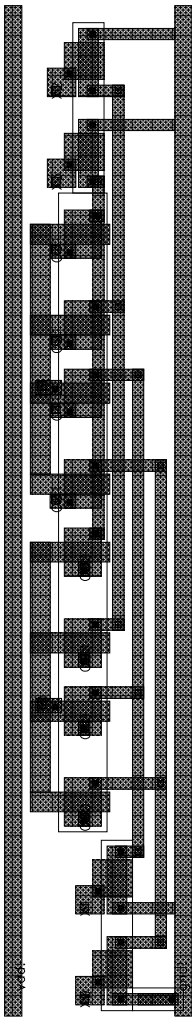

Figure C.15. Layout of the product_4_2 cell.

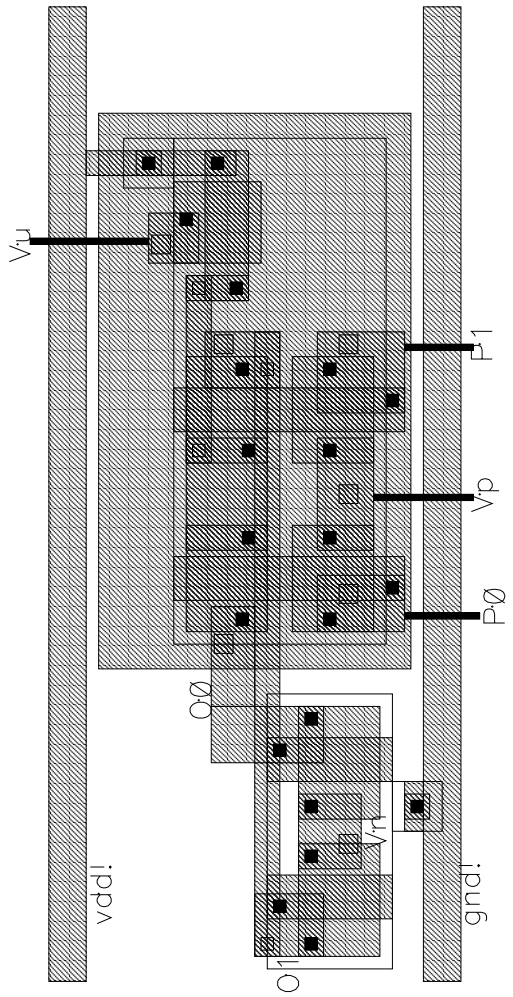

Figure C.16. Layout of the norm2 cell.

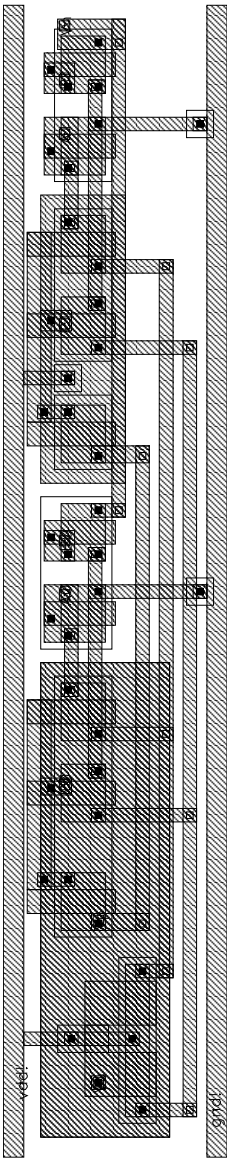

Figure C.17. Layout of the dnorm2 cell.

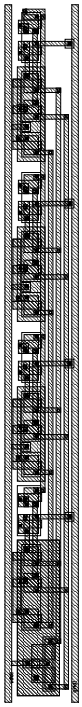

Figure C.18. Layout of the dnorm4 cell.

C.3 Interface of Cell Library

In order to do synthesis, two files decribing the interface of the cell library is needed as shown in Figure B.1. The description of analog_cell.vhd is shown below.

```
component PRODUCT_2_2
    port(X0 :  in real;
         X1 :  in real;
         Y0 :  in real;
         Y1 :  in real;
         O00 :  inout real;
         O01 :  inout real;
         O10 :  inout real;
         O11 :  inout real
        );
end component;
component PRODUCT_2_4
    port(X0 :  in real;
         X1 :  in real;
         Y0 :  in real;
         Y1 :  in real;
         Y2 :  in real;
         Y3 :  in real;
         O00 :  inout real;
         O01 :  inout real;
         O02 :  inout real;
         O03 :  inout real;
         O10 :  inout real;
         O11 :  inout real;
         O12 :  inout real;
         O13 :  inout real
        );
end component;
component PRODUCT_2_8
    port(X0 :  in real;
         X1 :  in real;
         Y0 :  in real;
         Y1 :  in real;
         Y2 :  in real;
         Y3 :  in real;
         Y4 :  in real;
         Y5 :  in real;
         Y6 :  in real;
         Y7 :  in real;
         O00 :  inout real;
         O01 :  inout real;
         O02 :  inout real;
         O03 :  inout real;
         O04 :  inout real;
         O05 :  inout real;
         O06 :  inout real;
         O07 :  inout real;
         O10 :  inout real;
         O11 :  inout real;
         O12 :  inout real;
         O13 :  inout real;
         O14 :  inout real;
         O15 :  inout real;
```

```
        O16 :  inout real;
        O17 :  inout real
       );
end component;
component PRODUCT_4_2
    port(X0 :  in real;
        X1 :  in real;
        X2 :  in real;
        X3 :  in real;
        Y0 :  in real;
        Y1 :  in real;
        O00 :  inout real;
        O01 :  inout real;
        O10 :  inout real;
        O11 :  inout real;
        O20 :  inout real;
        O21 :  inout real;
        O30 :  inout real;
        O31 :  inout real
       );
end component;
component PRODUCT_4_4
    port(X0 :  in real;
        X1 :  in real;
        X2 :  in real;
        X3 :  in real;
        Y0 :  in real;
        Y1 :  in real;
        Y2 :  in real;
        Y3 :  in real;
        O00 :  inout real;
        O01 :  inout real;
        O02 :  inout real;
        O03 :  inout real;
        O10 :  inout real;
        O11 :  inout real;
        O12 :  inout real;
        O13 :  inout real;
        O20 :  inout real;
        O21 :  inout real;
        O22 :  inout real;
        O23 :  inout real;
        O30 :  inout real;
        O31 :  inout real;
        O32 :  inout real;
        O33 :  inout real
       );
end component;
component PRODUCT_4_8
   port(X0 :  in real;
        X1 :  in real;
        X2 :  in real;
        X3 :  in real;
        Y0 :  in real;
        Y1 :  in real;
        Y2 :  in real;
        Y3 :  in real;
        Y4 :  in real;
```

```
        Y5 :  in real;
        Y6 :  in real;
        Y7 :  in real;
        O00 :  inout real;
        O01 :  inout real;
        O02 :  inout real;
        O03 :  inout real;
        O04 :  inout real;
        O05 :  inout real;
        O06 :  inout real;
        O07 :  inout real;
        O10 :  inout real;
        O11 :  inout real;
        O12 :  inout real;
        O13 :  inout real;
        O14 :  inout real;
        O15 :  inout real;
        O16 :  inout real;
        O17 :  inout real;
        O20 :  inout real;
        O21 :  inout real;
        O22 :  inout real;
        O23 :  inout real;
        O24 :  inout real;
        O25 :  inout real;
        O26 :  inout real;
        O27 :  inout real;
        O30 :  inout real;
        O31 :  inout real;
        O32 :  inout real;
        O33 :  inout real;
        O34 :  inout real;
        O35 :  inout real;
        O36 :  inout real;
        O37 :  inout real
       );
end component;
component PRODUCT_8_8
   port(X0 :  in real;
        X1 :  in real;
        X2 :  in real;
        X3 :  in real;
        X4 :  in real;
        X5 :  in real;
        X6 :  in real;
        X7 :  in real;
        Y0 :  in real;
        Y1 :  in real;
        Y2 :  in real;
        Y3 :  in real;
        Y4 :  in real;
        Y5 :  in real;
        Y6 :  in real;
        Y7 :  in real;
        O00 :  inout real;
        O01 :  inout real;
        O02 :  inout real;
        O03 :  inout real;
```

```
004 :  inout real;
005 :  inout real;
006 :  inout real;
007 :  inout real;
010 :  inout real;
011 :  inout real;
012 :  inout real;
013 :  inout real;
014 :  inout real;
015 :  inout real;
016 :  inout real;
017 :  inout real;
020 :  inout real;
021 :  inout real;
022 :  inout real;
023 :  inout real;
024 :  inout real;
025 :  inout real;
026 :  inout real;
027 :  inout real;
030 :  inout real;
031 :  inout real;
032 :  inout real;
033 :  inout real;
034 :  inout real;
035 :  inout real;
036 :  inout real;
037 :  inout real;
040 :  inout real;
041 :  inout real;
042 :  inout real;
043 :  inout real;
044 :  inout real;
045 :  inout real;
046 :  inout real;
047 :  inout real;
050 :  inout real;
051 :  inout real;
052 :  inout real;
053 :  inout real;
054 :  inout real;
055 :  inout real;
056 :  inout real;
057 :  inout real;
060 :  inout real;
061 :  inout real;
062 :  inout real;
063 :  inout real;
064 :  inout real;
065 :  inout real;
066 :  inout real;
067 :  inout real;
070 :  inout real;
071 :  inout real;
072 :  inout real;
073 :  inout real;
074 :  inout real;
075 :  inout real;
```

```
        O76 :  inout real;
        O77 :  inout real
       );
end component;
component NORM2
    port(Vu :  in real;
         Vp :  in real;
         Vn :  in real;
         P0 :  in real;
         P1 :  in real;
         O0 :  inout real;
         O1 :  inout real
         );
end component;
component NORM4
    port(Vu :  in real;
         Vp :  in real;
         Vn :  in real;
         P0 :  in real;
         P1 :  in real;
         P2 :  in real;
         P3 :  in real;
         O0 :  inout real;
         O1 :  inout real;
         O2 :  inout real;
         O3 :  inout real
         );
end component;
component NORM8
   port(Vu :  in real;
        Vp :  in real;
        Vn :  in real;
        P0 :  in real;
        P1 :  in real;
        P2 :  in real;
        P3 :  in real;
        P4 :  in real;
        P5 :  in real;
        P6 :  in real;
        P7 :  in real;
        X0 :  inout real;
        X1 :  inout real;
        X2 :  inout real;
        X3 :  inout real;
        X4 :  inout real;
        X5 :  inout real;
        X6 :  inout real;
        X7 :  inout real
         );
end component;
component DNORM2
    port(Vu :  in real;
         Vp :  in real;
         Vn :  in real;
         P0 :  in real;
         P1 :  in real;
         A0 :  inout real;
         A1 :  inout real;
```

```
        B0 :  inout real;
        B1 :  inout real
        );
end component;
component DNORM4
    port(Vu :  in real;
        Vp :  in real;
        Vn :  in real;
        P0 :  in real;
        P1 :  in real;
        P2 :  in real;
        P3 :  in real;
        A0 :  inout real;
        A1 :  inout real;
        A2 :  inout real;
        A3 :  inout real;
        B0 :  inout real;
        B1 :  inout real;
        B2 :  inout real;
        B3 :  inout real
        );
end component;
component DNORM8
   port(Vu :  in real;
       Vp :  in real;
       Vn :  in real;
       P0 :  in real;
       P1 :  in real;
       P2 :  in real;
       P3 :  in real;
       P4 :  in real;
       P5 :  in real;
       P6 :  in real;
       P7 :  in real;
       A0 :  inout real;
       A1 :  inout real;
       A2 :  inout real;
       A3 :  inout real;
       A4 :  inout real;
       A5 :  inout real;
       A6 :  inout real;
       A7 :  inout real;
       B0 :  inout real;
       B1 :  inout real;
       B2 :  inout real;
       B3 :  inout real;
       B4 :  inout real;
       B5 :  inout real;
       B6 :  inout real;
       B7 :  inout real
        );
end component;
```

The description of analog_cell.v is shown below.

```
//
// This file has interface descriptions for the analog
// cells.  Note that the cells have vdd!  and gnd!  as inputs to every
// symbol.  However, these power supply connections are treated as
```

```
// "special nets" in sedsm so they are NOT listed in the interface of
// the cells.  Note that in the LEF files the power supply ports are
// listed as "USE POWER" or "USE GROUND" to note this special usage.
//
// This file should be loaded into sedsm after loading the LEF file
// that describes the analog cell library.
// Use the "import verilog" command.  Only after this file is imported
// should you import the Verilog that has the description of your cell
// that you want to place and route.
//
// Jie Dai 03/14/02
//
module PRODUCT_2_2 (X0,X1,Y0,Y1,O00,O01,O10,O11);
   input X0,X1,Y0,Y1;
   inout O00,O01,O10,O11;
endmodule
module PRODUCT_2_4 (X0,X1,Y0,Y1,Y2,Y3,O00,O01,O02,O03,O10,O11,O12,O13);
   input X0,X1,Y0,Y1,Y2,Y3;
   inout O00,O01,O02,O03,O10,O11,O12,O13;
endmodule
module PRODUCT_2_8 (X0,X1,Y0,Y1,Y2,Y3,Y4,Y5,Y6,Y7,
                    O00,O01,O02,O03,O04,O05,O06,O07,O10,O11,O12,O13,O14,O15,O16,O17);
   input X0,X1,Y0,Y1,Y2,Y3,Y4,Y5,Y6,Y7;
   inout O00,O01,O02,O03,O04,O05,O06,O07,O10,O11,O12,O13,O14,O15,O16,O17;
endmodule
module PRODUCT_4_2 (X0,X1,X2,X3,Y0,Y1,O00,O01,O10,O11,O20,O21,O30,O31);
   input X0,X1,X2,X3,Y0,Y1;
   inout O00,O01,O10,O11,O20,O21,O30,O31;
endmodule
module PRODUCT_4_4 (X0,X1,X2,X3,Y0,Y1,Y2,Y3,
                    O00,O01,O02,O03,O10,O11,O12,O13,O20,O21,O22,O23,O30,O31,O32,O33);
   input X0,X1,X2,X3,Y0,Y1,Y2,Y3;
   inout O00,O01,O02,O03,O10,O11,O12,O13,O20,O21,O22,O23,O30,O31,O32,O33;
endmodule
module PRODUCT_4_8 (X0,X1,X2,X3,Y0,Y1,Y2,Y3,Y4,Y5,Y6,Y7,
                    O00,O01,O02,O03,O04,O05,O06,O07,O10,O11,O12,O13,O14,O15,O16,O17,
                    O20,O21,O22,O23,O24,O25,O26,O27,O30,O31,O32,O33,O34,O35,O36,O37);
   input X0,X1,X2,X3,Y0,Y1,Y2,Y3,Y4,Y5,Y6,Y7;
   inout O00,O01,O02,O03,O04,O05,O06,O07,O10,O11,O12,O13,O14,O15,O16,O17,
         O20,O21,O22,O23,O24,O25,O26,O27,O30,O31,O32,O33,O34,O35,O36,O37;
endmodule
module PRODUCT_8_8 (X0,X1,X2,X3,X4,X5,X6,X7,Y0,Y1,Y2,Y3,Y4,Y5,Y6,Y7,
                    O00,O01,O02,O03,O04,O05,O06,O07,O10,O11,O12,O13,O14,O15,O16,O17,
                    O20,O21,O22,O23,O24,O25,O26,O27,O30,O31,O32,O33,O34,O35,O36,O37,
                    O40,O41,O42,O43,O44,O45,O46,O47,O50,O51,O52,O53,O54,O55,O56,O57,
                    O60,O61,O62,O63,O64,O65,O66,O67,O70,O71,O72,O73,O74,O75,O76,O77);
   input X0,X1,X2,X3,X4,X5,X6,X7,Y0,Y1,Y2,Y3,Y4,Y5,Y6,Y7;
   inout O00,O01,O02,O03,O04,O05,O06,O07,O10,O11,O12,O13,O14,O15,O16,O17,
         O20,O21,O22,O23,O24,O25,O26,O27,O30,O31,O32,O33,O34,O35,O36,O37,
         O40,O41,O42,O43,O44,O45,O46,O47,O50,O51,O52,O53,O54,O55,O56,O57,
         O60,O61,O62,O63,O64,O65,O66,O67,O70,O71,O72,O73,O74,O75,O76,O77;
endmodule
module NORM2 (Vu,Vp,Vn,P0,P1,O0,O1);
   input Vu,Vp,Vn,P0,P1;
   output O0,O1;
endmodule
module NORM4 (Vu,Vp,Vn,P0,P1,P2,P3,O0,O1,O2,O3);
   input Vu,Vp,Vn,P0,P1,P2,P3;
```

```
   output O0,O1,O2,O3;
endmodule
module NORM8 (Vu,Vp,Vn,P0,P1,P2,P3,P4,P5,P6,P7,O0,O1,O2,O3,O4,O5,O6,O7);
   input Vu,Vp,Vn,P0,P1,P2,P3,P4,P5,P6,P7;
   output O0,O1,O2,O3,O4,O5,O6,O7;
endmodule
module DNORM2 (Vu,Vp,Vn,P0,P1,A0,A1,B0,B1);
   input Vu,Vp,Vn,P0,P1;
   output A0,A1,B0,B1;
endmodule
module DNORM4 (Vu,Vp,Vn,P0,P1,P2,P3,A0,A1,A2,A3,B0,B1,B2,B3);
   input Vu,Vp,Vn,P0,P1,P2,P3;
   output A0,A1,A2,A3,B0,B1,B2,B3;
endmodule
module DNORM8 (Vu,Vp,Vn,P0,P1,P2,P3,P4,P5,P6,P7,A0,A1,A2,A3,A4,A5,A6,A7,
               B0,B1,B2,B3,B4,B5,B6,B7);
   input Vu,Vp,Vn,P0,P1,P2,P3,P4,P5,P6,P7;
   output A0,A1,A2,A3,A4,A5,A6,A7,B0,B1,B2,B3,B4,B5,B6,B7;
endmodule
```

C.4 VHDL Description Examples

In this section, some VHDL description examples are provided.

C.4.1 VHDL Description of the Equal Gate Function Node and XOR Function Node

In this subsection, the high-level behavioral description of the equal gate function node and XOR function node with connection to three variable nodes x, y and z shown in Figure 3.14 is provided below.

Equal gate:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.math_real.all;
type real_array_1d is array ( natural range <> )
of real;
entity xor3 is
  port(clk :  in std_logic;
       xi :  in real_array_1d(0 to 1);
       yi :  in real_array_1d(0 to 1);
       zi :  in real_array_1d(0 to 1);
       xo :  out real_array_1d(0 to 1) :=(0.5,0.5);
       yo :  out real_array_1d(0 to 1) :=(0.5,0.5);
       zo :  out real_array_1d(0 to 1) :=(0.5,0.5)
       );
end xor3;
architecture behavior of xor3 is
begin
  xor3:process(clk)
    variable temp_xo :  real_array_1d(0 to 1);
    variable temp_x :  real;
```

```
    variable temp_yo :  real_array_1d(0 to 1);
    variable temp_y :  real;
    variable temp_zo :  real_array_1d(0 to 1);
    variable temp_z :  real;
    if(clk='1') then
      temp_xo(0):=yi(0)*zi(0);
      temp_xo(1):=yi(1)*zi(1);
      temp_x:=temp_xo(0)+temp_xo(1);
      xo(0)<=temp_xo(0)/temp_x;
      xo(1)<=temp_xo(1)/temp_x;
      temp_yo(0):=xi(0)*zi(0);
      temp_yo(1):=xi(1)*zi(1);
      temp_y:=temp_yo(0)+temp_yo(1);
      yo(0)<=temp_yo(0)/temp_y;
      yo(1)<=temp_yo(1)/temp_y;
      temp_zo(0):=xi(0)*yi(0);
      temp_zo(1):=xi(1)*yi(1);
      temp_z:=temp_zo(0)+temp_zo(1);
      zo(0)<=temp_zo(0)/temp_z;
      zo(1)<=temp_zo(1)/temp_z;
    end if;
  end process;
end behavior;
```

XOR function node:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.math_real.all;
type real_array_1d is array ( natural range <> )
of real;
entity xor3 is
  port(clk :  in std_logic;
       xi :  in real_array_1d(0 to 1);
       yi :  in real_array_1d(0 to 1);
       zi :  in real_array_1d(0 to 1);
       xo :  out real_array_1d(0 to 1) :=(0.5,0.5);
       yo :  out real_array_1d(0 to 1) :=(0.5,0.5);
       zo :  out real_array_1d(0 to 1) :=(0.5,0.5)
       );
end xor3;
architecture behavior of xor3 is
begin
  xor3:process(clk)
    variable temp_xo :  real_array_1d(0 to 1);
    variable temp_x :  real;
    variable temp_yo :  real_array_1d(0 to 1);
    variable temp_y :  real;
    variable temp_zo :  real_array_1d(0 to 1);
    variable temp_z :  real;
    if(clk='1') then
      temp_xo(0):=yi(0)*zi(0)+yi(1)*zi(1);
      temp_xo(1):=yi(0)*zi(1)+yi(1)*zi(0);
      temp_x:=temp_xo(0)+temp_xo(1);
      xo(0)<=temp_xo(0)/temp_x;
      xo(1)<=temp_xo(1)/temp_x;
```

```
      temp_yo(0):=xi(0)*zi(0)+xi(1)*zi(1);
      temp_yo(1):=xi(0)*zi(1)+xi(1)*zi(0);
      temp_y:=temp_yo(0)+temp_yo(1);
      yo(0)<=temp_yo(0)/temp_y;
      yo(1)<=temp_yo(1)/temp_y;
      temp_zo(0):=xi(0)*yi(0)+xi(1)*yi(1);
      temp_zo(1):=xi(0)*yi(1)+xi(1)*yi(0);
      temp_z:=temp_zo(0)+temp_zo(1);
      zo(0)<=temp_zo(0)/temp_z;
      zo(1)<=temp_zo(1)/temp_z;
    end if;
  end process;
end behavior;
```

C.4.2 Automatically Generated VHDL Files for the Hamming (8,4) Decoder

In this subsection, the automatically generated VHDL files for the high-level simulation of the Hamming (8,4) decoder using the factor graph description provided in Appendix A.5 are provided below.

```
package types is
type bin_enum is (e_0,e_1);
type bin is array (bin_enum) of real;
type bin_array_1d is array (natural range <>) of bin;
type bin_array_2d is array (natural range <>,natural range<>) of bin;
type quat_enum is (e_00,e_01,e_10,e_11);
type quat is array (quat_enum) of real;
type quat_array_1d is array (natural range <>) of quat;
type quat_array_2d is array (natural range <>,natural range<>) of quat;
type state4_enum is (e_0,e_1,e_2,e_3);
type state4 is array (state4_enum) of real;
type state4_array_1d is array (natural range <>) of state4;
type state4_array_2d is array (natural range <>,natural range<>) of state4;
end types;
library ieee;
use ieee.std_logic_1164.all;
use ieee.math_real.all;
use work.types.all;
use work.myarray.all;
use work.nondeterminism.all;
use work.gaussian.all;
entity fa is
port(clk :  in std_logic := '0';
cs :  in bin := (others=>0.5);
cs_out :  out bin := (others=>0.5);
u_out :  out bin := (others=>0.5);
x :  in quat := (others=>0.25);
ns :  in state4 := (others=>0.25);
ns_out :  out state4 := (others=>0.25));
end fa;
architecture structure of fa is
begin
fa:process(clk)
variable temp_cs_out :  bin;
```

```
variable temp_cs :  real;
variable temp_u_out :  bin;
variable temp_u :  real;
variable temp_ns_out :  state4;
variable temp_ns :  real;
begin
if(clk = '1') then
temp_cs_out(e_0) := x(e_00) * ns(e_0) + x(e_11) * ns(e_1);
temp_cs_out(e_1) := x(e_01) * ns(e_2) + x(e_10) * ns(e_3);
temp_cs := temp_cs_out(e_0) + temp_cs_out(e_1);
cs_out(e_0) <= temp_cs_out(e_0) / temp_cs;
cs_out(e_1) <= temp_cs_out(e_1) / temp_cs;
temp_u_out(e_0) := cs(e_0) * x(e_00) * ns(e_0) + cs(e_1) * x(e_01) * ns(e_2);
temp_u_out(e_1) := cs(e_0) * x(e_11) * ns(e_1) + cs(e_1) * x(e_10) * ns(e_3);
temp_u := temp_u_out(e_0) + temp_u_out(e_1);
u_out(e_0) <= temp_u_out(e_0) / temp_u;
u_out(e_1) <= temp_u_out(e_1) / temp_u;
temp_ns_out(e_0) := cs(e_0) * x(e_00);
temp_ns_out(e_1) := cs(e_0) * x(e_11);
temp_ns_out(e_2) := cs(e_1) * x(e_01);
temp_ns_out(e_3) := cs(e_1) * x(e_10);
temp_ns := temp_ns_out(e_0) + temp_ns_out(e_1) + temp_ns_out(e_2) + temp_ns_out(e_3);
ns_out(e_0) <= temp_ns_out(e_0) / temp_ns;
ns_out(e_1) <= temp_ns_out(e_1) / temp_ns;
ns_out(e_2) <= temp_ns_out(e_2) / temp_ns;
ns_out(e_3) <= temp_ns_out(e_3) / temp_ns;
end if;
end process;
end structure;
library ieee;
use ieee.std_logic_1164.all;
use ieee.math_real.all;
use work.types.all;
use work.myarray.all;
use work.nondeterminism.all;
use work.gaussian.all;
entity fb is
port(clk :  in std_logic := '0';
cs :  in state4 := (others=>0.25);
cs_out :  out state4 := (others=>0.25);
u_out :  out bin := (others=>0.5);
x :  in quat := (others=>0.25);
ns :  in bin := (others=>0.5);
ns_out :  out bin := (others=>0.5));
end fb;
architecture structure of fb is
begin
fb:process(clk)
variable temp_cs_out :  state4;
variable temp_cs :  real;
variable temp_u_out :  bin;
variable temp_u :  real;
variable temp_ns_out :  bin;
variable temp_ns :  real;
begin
if(clk = '1') then
temp_cs_out(e_0) := x(e_00) * ns(e_0) + x(e_11) * ns(e_1);
temp_cs_out(e_1) := x(e_11) * ns(e_0) + x(e_00) * ns(e_1);
```

```
temp_cs_out(e_2) := x(e_10) * ns(e_0) + x(e_01) * ns(e_1);
temp_cs_out(e_3) := x(e_01) * ns(e_0) + x(e_10) * ns(e_1);
temp_cs := temp_cs_out(e_0) + temp_cs_out(e_1) + temp_cs_out(e_2) + temp_cs_out(e_3);
cs_out(e_0) <= temp_cs_out(e_0) / temp_cs;
cs_out(e_1) <= temp_cs_out(e_1) / temp_cs;
cs_out(e_2) <= temp_cs_out(e_2) / temp_cs;
cs_out(e_3) <= temp_cs_out(e_3) / temp_cs;
temp_u_out(e_0) := cs(e_0) * x(e_00) * ns(e_0) + cs(e_1) * x(e_11) * ns(e_0)
                 + cs(e_2) * x(e_10) * ns(e_0) + cs(e_3) * x(e_01) * ns(e_0);
temp_u_out(e_1) := cs(e_0) * x(e_11) * ns(e_1) + cs(e_1) * x(e_00) * ns(e_1)
                 + cs(e_2) * x(e_01) * ns(e_1) + cs(e_3) * x(e_10) * ns(e_1);
temp_u := temp_u_out(e_0) + temp_u_out(e_1);
u_out(e_0) <= temp_u_out(e_0) / temp_u;
u_out(e_1) <= temp_u_out(e_1) / temp_u;
temp_ns_out(e_0) := cs(e_0) * x(e_00) + cs(e_1) * x(e_11)
                 + cs(e_2) * x(e_10) + cs(e_3) * x(e_01);
temp_ns_out(e_1) := cs(e_0) * x(e_11) + cs(e_1) * x(e_00)
                 + cs(e_2) * x(e_01) + cs(e_3) * x(e_10);
temp_ns := temp_ns_out(e_0) + temp_ns_out(e_1);
ns_out(e_0) <= temp_ns_out(e_0) / temp_ns;
ns_out(e_1) <= temp_ns_out(e_1) / temp_ns;
end if;
end process;
end structure;
library ieee;
use ieee.std_logic_1164.all;
use ieee.math_real.all;
use work.types.all;
use work.myarray.all;
use work.nondeterminism.all;
use work.gaussian.all;
entity pdf is
port(clk :  in std_logic := '0';
n0 :  in real := 0.0;
y :  in real_array_1d(1 to 2) := (others=>0.0);
x_out :  out quat := (others=>0.25));
end pdf;
architecture structure of pdf is
begin
pdf:process(clk)
variable p0 :  real := 0.0;
variable p1 :  real := 0.0;
begin
if(clk = '1') then
p0:=1.0/(1.0+exp(-4.0*y(1)/n0));
p1:=1.0/(1.0+exp(-4.0*y(2)/n0));
x_out(e_00)<=(1.0-p0)*(1.0-p1);
x_out(e_01)<=(1.0-p0)*p1;
x_out(e_10)<=p0*(1.0-p1);
x_out(e_11)<=p0*p1;
end if;
end process;
end structure;
library ieee;
use ieee.std_logic_1164.all;
use ieee.math_real.all;
use work.types.all;
use work.myarray.all;
```

```
use work.nondeterminism.all;
use work.gaussian.all;
entity CodeGraph is
port(clk :  in std_logic := '0';
x :  in quat_array_1d(1 to 4) := (others=>(others=>0.25));
u_out :  out bin_array_1d(1 to 4) := (others=>(others=>0.5)));
end CodeGraph;
architecture structure of CodeGraph is
signal s0 :  bin;
signal s0_out :  bin;
signal s1 :  state4;
signal s1_out :  state4;
signal s2 :  bin;
signal s2_out :  bin;
signal s3 :  state4;
signal s3_out :  state4;
begin
f0 :  entity work.fa(structure) port map(clk=>clk, cs=>s0, cs_out=>s0_out,
                                         u_out=>u_out(1), x=>x(1),
                                         ns=>s1, ns_out=>s1_out);
f1 :  entity work.fb(structure) port map(clk=>clk, cs=>s1_out, cs_out=>s1,
                                         u_out=>u_out(2), x=>x(2),
                                         ns=>s2, ns_out=>s2_out);
f2 :  entity work.fa(structure) port map(clk=>clk, cs=>s2_out, cs_out=>s2,
                                         u_out=>u_out(3), x=>x(3),
                                         ns=>s3, ns_out=>s3_out);
f3 :  entity work.fb(structure) port map(clk=>clk, cs=>s3_out, cs_out=>s3,
                                         u_out=>u_out(4), x=>x(4),
                                         ns=>s0_out, ns_out=>s0);
end structure;
library ieee;
use ieee.std_logic_1164.all;
use ieee.math_real.all;
use work.types.all;
use work.myarray.all;
use work.nondeterminism.all;
use work.gaussian.all;
entity decoder is
port(clk :  in std_logic := '0';
n0 :  in real := 0.0;
y :  in real_array_1d(1 to 8) := (others=>0.0);
u_out :  out bin_array_1d(1 to 4) := (others=>(others=>0.5)));
end decoder;
architecture structure of decoder is
signal x :  quat_array_1d(1 to 4);
signal x_out :  quat_array_1d(1 to 4);
begin
g0:  for i in 1 to 4 generate
begin
pdfi :  entity work.pdf(structure) port map(clk=>clk,
                                            n0=>n0,
                                            y=>y(2*i-1 to 2*i),
                                            x_out=>x_out(i));
end generate g0;
TheCodeGraph :  entity work.CodeGraph(structure) port map(clk=>clk,
                                                          x=>x_out(1 to 4),
                                                          u_out=>u_out(1 to 4));
end structure;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.math_real.all;
use work.types.all;
use work.myarray.all;
use work.nondeterminism.all;
use work.gaussian.all;
package genfunction is
function gen(s :  real_array_1d) return real_array_1d;
end genfunction;
package body genfunction is
function gen(s :  real_array_1d) return real_array_1d is
variable x :  real_array_1d(1 to 8);
variable p0 :  real;
variable p1 :  real;
variable genarray :  real_array_2d(1 to 4,1 to 8);
begin
genarray:=((1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0),
           (0.0, 0.0, 1.0, 1.0, 0.0, 1.0, 1.0, 0.0),
           (0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0),
           (0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 1.0, 1.0));
x:=s*genarray mod 2.0;
return x;
end gen;
end genfunction;
library ieee;
use ieee.std_logic_1164.all;
use ieee.math_real.all;
use work.types.all;
use work.myarray.all;
use work.nondeterminism.all;
use work.gaussian.all;
package comp is
function convert(u :  bin_array_1d) return real_array_1d;
function compare(l :  real_array_1d; r :  real_array_1d) return real;
end comp;
package body comp is
function convert(u :  bin_array_1d) return real_array_1d is
variable result :  real_array_1d(1 to 4);
begin
for i in u'range loop
if (u(i)(e_0) >= u(i)(e_1)) then
result(i) := 0.0;
else
result(i) := 1.0;
end if;
end loop;
return result;
end convert;
function compare(l :  real_array_1d; r :  real_array_1d) return real is
variable result :  real := 0.0;
begin
for i in l'range loop
if(l(i) /= r(i)) then
result := result + 1.0;
end if;
end loop;
return result;
```

```
end compare;
end comp;
library ieee;
use ieee.std_logic_1164.all;
use ieee.math_real.all;
use work.types.all;
use work.myarray.all;
use work.nondeterminism.all;
use work.gaussian.all;
use work.genfunction.all;
use work.comp.all;
use std.textio.all;
entity encoder is
generic(MaxSNR: real; frame_length :  real;rate :  real; cycles :  real);
port(clk :  in std_logic := '0';
n0 :  inout real := 1.0/rate;
u :  in bin_array_1d(1 to 4) := (others=>(others=>0.5));
y_out :  out real_array_1d(1 to 8) := (others=>0.0));
end encoder;
architecture structure of encoder is
file ber_file:  text open write_mode is "ber.dat";
begin
encoder:process
variable snr :  real := 0.0;
variable error_number :  real := 0.0;
variable clk_count :  real := 0.0;
variable frame_count :  real := -1.0;
variable frame_error :  real := 0.0;
variable e :  real;
variable li :  line;
variable p0 :  real := 0.0;
variable p1 :  real := 0.0;
variable genarray :  real_array_2d(1 to 4,1 to 8) := (others=>(others=>0.0));
variable s :  real_array_1d(1 to 4) := (others=>0.0);
variable olds :  real_array_1d(1 to 4) := (others=>0.0);
begin
wait on clk;
if(clk = '1') then
if(clk_count=0.0) then
olds:=s;
s:=rand(4);
y_out<=2.0*gen(s)-1.0+randn(sqrt(n0/2.0),8);
frame_count := frame_count + 1.0;
if(frame_count/=0.0) then
e := compare(convert(u),olds);
error_number := error_number + e;
if(e /= 0.0) then
frame_error := frame_error + 1.0;
end if;
end if;
if(frame_error >= 150.0) then
write(li,snr);
write(li,',');
write(li,error_number/(frame_count*real(frame_length)));
write(li,CR);
writeline(ber_file,li);
snr := snr + 1.0;
if(snr > MaxSNR) then
```

```
wait;
end if;
frame_count := -1.0;
error_number := 0.0;
frame_error := 0.0;
n0<= 1.0 / (rate*10.0**(snr/10.0));
end if;
end if;
clk_count := clk_count + 1.0;
if(clk_count = cycles) then
clk_count := 0.0;
end if;
end if;
end process;
end structure;
library ieee;
use ieee.std_logic_1164.all;
use ieee.math_real.all;
use work.types.all;
use work.myarray.all;
use work.nondeterminism.all;
use work.gaussian.all;
entity top is
end top;
architecture structure of top is
signal clk :  std_logic;
signal n0 :  real;
signal n0_out :  real;
signal u :  bin_array_1d(1 to 4);
signal u_out :  bin_array_1d(1 to 4);
signal y :  real_array_1d(1 to 8);
signal y_out :  real_array_1d(1 to 8);
begin
en :  entity work.encoder(structure) generic map(7.0, 4.0, 0.5, 20.0)
                                     port map(clk=>clk, n0=>n0, u=>u, y_out=>y);
de :  entity work.decoder(structure) port map(clk=>clk, n0=>n0, y=>y, u_out=>u);
TheClk :  entity work.clk_gen(structure) port map(clk=>clk);
end structure;
```

REFERENCES

[1] Aji, A. M., Horn, G. B., and McEliece, R. J. Iterative decoding on graphs with a single cycle. In *Proceedings of the IEEE international Symposium on Information Theory* (Aug. 1998), p. 276.

[2] Aji, S. M., and McEliece, R. J. The generalized distributive law. *IEEE Trans. Inform. Theory* (Mar. 2000), 325–343.

[3] Anderson, J. B., and Hladik, S. M. Tailbiting map decoders. *IEEE Journal on Selected Areas in Communications 16*, 2 (Feb. 1998), 297–302.

[4] Bahl, L. R., Cocke, J., Jelinek, F., and Raviv, J. Optimal decoding of linear codes for minimizing symbol error rate. *IEEE Trans. Inform. Theory* (March 1974), 284–287.

[5] Bastos, J., Steyaert, M., Pergoot, A., and Sansen, W. Mismatch characterization of submicron mos transistors. *Analog Integerated Circuits and Signal Processing 12* (1997), 95–106.

[6] Benedetto, S., Divsalar, D., Montorsi, G., and Pollara, F. Serial concatenation of interleaved codes: Performance analysis, design and iterative decoding. *IEEE Transactions on Information Theory 44*, 3 (May 1998), 909–926.

[7] Benedetto, S., and montorsi, G. Iterative decoding of serially concatenated convolutional codes. *Electronics Letters 32* (June 1996), 1186–1188.

[8] Benedetto, S., and Montorsi, G. Unveiling turbo codes: some results on parallel concatenated coding schemes. *IEEE Transactions on Information Theory 42* (Mar. 1996), 409–428.

[9] Berrou, C., Glavieux, A., and Thitimajshima, P. Near shannon limit error-correcting coding and decoding: turbo codes. In *Proceedings of the International Conference on Communications* (May 1993), pp. 1064–1070.

[10] Calderbank, A. R., Forney, G. D., and Vardy, A. Minimal tail-biting trellises: The golay code and more. *IEEE Trans. Inform. Theory* (1999), 1435–1455.

[11] Damen, M. O., Tewfik, A., and Belfiore, J.-C. A construction of a space-time code based on number theory. *IEEE Transactions on Information Theory 48*, 3 (Mar. 2002), 753–760.

[12] Divsalar, D., Jin, H., and McEliece, R. J. Coding theorems for 'turbo-like' codes. In *Proceedings of 36th Allerton Conference on Communication, Control, and Computing* (Sept. 1998), pp. 201–210.

[13] Drager, S. L., Carter, H. W., and Hirsch, H. L. A vhdl-ams mixed-signal, mixed-technology design tool. In *Proc. of the IEEE 1998 National Aerospace and Electronics Conference* (1998).

[14] El-Turky, F., and Perry, E. Blades: An artificial intelligence approach to analog circuit design. *IEEE Transactions on Computer-Aided Design 8* (June 1989), 680–691.

[15] Enz, C. C., Krummenacher, F., and Vittoz, E. A. A analytical mos transistor model valid in all regions of operation and dedicated to low-voltage and low-current applications. *Analog Integrated Circuits and Signal Processing 8* (July 1995), 83–114.

[16] Ferrari, M., and Bellini, S. Importance sampling simulation of concatenated block codes. In *IEE Proceedings-Communications* (Oct. 2000), vol. 147, pp. 245–251.

[17] Frey, B. J., Kschischang, F. R., Loeliger, H. A., and Wiberg, N. Factor graphs and algorithms. In *Proc. 35th Allerton Conf. on Communications, Control, and Computing* (Sept. 1997), pp. 666–680.

[18] G. D. Forney, J. The viterbi algorithm. *Proceedings of the IEEE 61*, 3 (Mar. 1973), 268–278.

[19] Gallager, R. G. *Low-Density Parity-Check Codes*. MIT press, 1963.

[20] Gielen, G. E., and Rutenbar, R. A. Computer-aided design of analog and mixed-signal integrated circuits. In *Proceedings of the IEEE* (Dec. 1989), vol. 88, pp. 1823–1852.

[21] Gilbert, B. Translinear circuits: A proposed classification. *Electronics Letters 11*, 1 (Jan. 1975), 14–16.

[22] Gilbert, B. A monolithic 16-channel analog array normalizer. *IEEE Journal of Solid-State Circuits 19* (1984), 956–963.

[23] Gilbert, B. Translinear circuits: An historical overview. *Analog Integrated Circuits and Signal Processing 9*, 2 (Mar. 1996), 95–118.

[24] Godfrey, M. D. Cmos device modeling for subthreshold circuits. *IEEE Trans. Circuits and Systems* (Aug. 1992), 532–539.

[25] Hagenauer, J., Offer, E., and Papke, L. Iterative decoding of binary block and convolutional codes. *IEEE Transactions on Information Theory 42*, 2 (Mar. 1996), 429–445.

[26] Hagenauer, J., and Winkelhofer, M. The analog decoder. In *IEEE International Symposium on Information Theory* (Aug. 1998), pp. 145–145.

[27] Hamming, R. W. Error detecting and error correcting codes. *Bell Systems Technical Journal 29* (1950), 147–160.

[28] Harjani, R., Rutenbar, R., and Carley, L. R. Oasys: A framework for analog circuit synthesis. *IEEE Transactions on Computer-Aided Design 8* (Dec. 1989), 1247–1265.

[29] Johns, D. A., and Martin, K. *Analog Integrated Circuit Design*. John Wiley & Sons, 1997.

[30] Jr., G. D. F. Codes on graphs: Normal realizations. In *Proceedings of the IEEE International Symposium on Information Theory* (2000), pp. 9–9.

[31] Jr., G. D. F. Codes on graphs: Normal realizations. *IEEE Transactions on Information Theory 47*, 2 (Feb. 2001), 520–548.

[32] Kinget, P., and Steyaert, M. Impact of transistor mismatch on the speed-accuracy-power trade-off of analog cmos circuits. In *IEEE 1996 Custom Integrated Circuits Conference* (1996), pp. 333–336.

[33] Koh, H., Séquin, C., and Gray, P. Opasyn: A compiler for cmos operational amplifiers. *IEEE Transactions on Computer-Aided Design 9* (Feb. 1990), 113–125.

[34] Kschischang, F. R., and Frey, B. J. Iterative decoding of compound codes by probability propagation in graphical models. *IEEE Journal on Selected Areas in Communications 16*, 2 (Feb. 1998), 219–230.

[35] Kschischang, F. R., Frey, B. J., and Loeliger, H. A. Factor graphs and the sum-product algorithm. *IEEE Trans. Inform. Theory* (Feb. 2001), 498–519.

[36] Lakshmikumar, K. R., Hadaway, R. A., and Copeland, M. A. Characterization and modeling of mismatch in mos transistors for precision analog design. *IEEE Jouranl of Solid-State Circuits 21* (Dec. 1986), 1057–1066.

[37] Lin, S., and D. J. Costello, J. *Error Control Coding: Fundamentals and Applications*. Prentice Hall Series in Computer Applications in Electrical Engineering. Prentice Hall, Englewood Cliffs, NJ, 1983.

[38] Loeliger, H. A., Lustenberger, F., Helfenstein, M., and Tarköy, F. Probability propagation and decoding in analog vlsi. In *IEEE Int. Symp. on Information Theory* (Aug. 1998), pp. 146–146.

[39] Loeliger, H.-A., Lustenberger, F., Helfenstein, M., and Tarköy, F. Probability propagation and decoding in analog vlsi. *IEEE Transcations on Information Theory 47*, 2 (Feb. 2001), 837–843.

[40] Loeliger, H.-A., Lustenberger, F., Tarköy, F., and Helfenstein, M. Decoding in analog vlsi. *IEEE Communications Magazine 37*, 4 (Apr. 1999), 99–101.

[41] Luby, M. G., Mitzenmacher, M., Shokrollahi, M. A., and Spielmann, D. A. Improved low-density parity-check codes using irregular graphs and belief propagation. In *Proceedings of the IEEE International Symposium on Information Theory* (Aug. 1998), pp. 117–117.

[42] Lustenberger, F. *On the Design of Analog VLSI Iterative Decoders*. PhD thesis, Swiss Federal Institute of Technology, 2000.

[43] Lustenberger, F., Helfenstein, M., Loeliger, H.-A., Tarköy, F., and Moschytz, G. S. All-analog decoder for a binary (18,9,5) tail-biting trellis code. In *Proceedings of the European Solid-State Circuits Conference* (Sept. 1999), pp. 362–365.

[44] Lustenberger, F., Helfenstein, M., Loeliger, H.-A., Tarköy, F., and Moschytz, G. S. An analog vlsi decoding technique for digital codes. In *Proceedings of the IEEE International Symposium on Circuits and Systems* (May / June 1999), vol. 2, pp. 424–427.

[45] Lustenberger, F., and Loeliger, H. A. On mismatch errors in analog-vlsi error correcting decoders. In *Proceedings of ISCAS* (May 2001).

[46] MacKay, D. J. C. Good error-correcting codes based on very sparse matrices. *IEEE Transactions on Information Theory 45*, 2 (Mar. 1999), 399–431.

[47] Mackay, D. J. C., and Neal, R. M. Good codes based on very sparse matrices. In *Cryptography and Coding. Fifth IMA Conference* (1995), pp. 100–111.

[48] MacWilliams, F. J., and Sloane, N. J. A. *The Theory of Error Correcting Codes*. North-Holland, 1977.

[49] Martin, P. A., and Taylor, D. P. On adaptive reduced-complexity iterative decoding. In *Globecom '00 - IEEE. Global Telecommunications Conference* (San Francison, CA, Nov. / Dec. 2000), vol. 2, pp. 732–737.

[50] Massey, J. L. Foundations and methods of channel coding. In *Proceedings of International Conference on Information Theory Systems* (1978), vol. 65, pp. 148–157.

[51] Maulik, P., Carley, L. R., and Rutenbar, R. Simultaneous topology selection and sizing of cell-level analog circuits. *IEEE Transactions on Computer-Aided Design 14* (Apr. 1995), 401–412.

[52] Mead, C. A. *Analog VLSI and Neural Systems*. Addison Wesley Computation and Neural Systems Series. Addison Wesley, MA, 1989.

[53] Michael, C., and M.Ismail. Statistical modeling of device mismatch for analog mos integrated circuits. *IEEE Jouranl of Solid-State Circuits 27* (Feb. 1992), 154–166.

[54] Moerz, M., Gabara, T., Yan, R., and Hagenauer, J. An analog 0.25um bicmos tailbiting map decoder. In *IEEE Proc. International Solid-State Circuits Conference* (Feb. 2000), pp. 356–357.

[55] Pelgrom, M. J., Duinmaijer, A. C. J., and Welbers, A. P. G. Matching properties of mos transistors. *IEEE Journal of Solid-State Circuits 24*, 5 (Oct. 1989), 1433–1439.

[56] Razavi, B. *Design of Analog CMOS Integrated Circuits*. McGraw Hill, 2001.

[57] Richardson, T., Shokrollahi, A., and Urbanke, R. Design of provable good low-density parity check codes. *IEEE Transactions on Information Theory* (Apr. 1999).

[58] Schlegel, C., and Grant, A. Concatenated space-time coding. In *Proc. Symposium on Personal, Indoor and Mobile Radio Communications, PIMRC 2001* (Sept./Oct. 2001), pp. 139–143.

[59] Schlegel, C., and Grant, A. Differential turbo space-time coding. In *Proceedings of 2001 IEEE Information Theory Workshop* (Sept. 2001), pp. 120–122.

[60] Seevinck, E. *Analysis and Synthesis of Translinear Integrated Circuits*. Amsterdam, 1988.

[61] Shannon, C. E. A mathematical theory of communication. *Bell Systems Technical Journal 27* (July 1948), 379–423.

[62] Sweeney, P. *Error Control Coding: An Introduction*. Prentice Hall, 1991.

[63] Tanner, R. M. A recursive approach to low complexity codes. *IEEE Transactions on Information Theory 27*, 5 (1981), 533–547.

[64] ten Brink, S. Convergence of iterative decoding. *Electronics Letters 35*, 13 (June 1999), 1117–1119.

[65] Tsividis, Y. *Operation and modeling of the MOS transistor*. 1999.

[66] Viterbi, A. J. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory 13* (Apr. 1967), 260–269.

[67] Wiberg, N., Loeliger, H. A., and Kotter, R. Codes and iterative decoding on general graphs. *European Transactions on Telecommunications* (Sept./Oct. 1995), 513–525.

[68] Willems, J. C. Models for dynamics. In *Dynamics Reported*, U. Kirchgraber and H. O. Walther, Eds. John Wiley and Sons, 1989, pp. 171–269.

[69] Winstead, C., Dai, J., Kim, W. J., Little, S., Kim, Y. B., Myers, C., and Schlegel, C. Analog map decoder for (8,4) hamming code in subthreshold cmos. In *Advanced Research in VLSI* (Mar. 2001), pp. 132–147.

[70] Winstead, C., Dai, J., Yu, S., Harrison, R., Myers, C., and Schlegel, C. Analog decoding of product codes. In *International Symposium on Information Technology* (June/July 2002).

[71] Wolf, J. K. Efficient maximum-likelihood decoding of linear block codes using a trellis. *IEEE Transactions on Information Theory 24*, 1 (1978), 76–80.

[72] Yu, N. Y., Kim, Y., and Lee, P. J. Iterativ decoding of product codes composed of extended hamming codes. In *IEEE Symopsium on Computers and Communications* (Antibes, France, July 2000), pp. 732–737.

[73] Yu, S. *Private Communications*, Fall 2001. Shuhuan Yu is a Ph.D. student at University of Utah.