School of Electronic Engineering Science

University of Wales, Bangor

Runlength Limited Error Control Codes: design, realisation and practical evaluation

Project dissertation

Carlos M. Mira da Fonseca

1991

Acknowlegement

The work presented in this dissertation was conducted in the course of a nine months period of studies at the University of Wales, Bangor, U.K., under the EEC ERASMUS programme. The author is grateful to Professor John O'Reilly (University of Wales) and Professor Manuel Duarte (University of Aveiro, Portugal) for establishing the present co-operation arrangement between the two Universities, and also to the ERASMUS programme and the University of Aveiro for financial support.

The author would like to express his special appreciation to Professor John O'Reilly (the project supervisor) for his guidance and advice throughout the project, and to Mr. S. Williams and Dr. A. Popplewell for the close support and assistance they always provided.

Abstract

The aim of this project is the practical realisation of a decoding scheme for runlength limited error control codes. Such codes have advantageous characteristics which make them particularly suitable for a number of data storage and transmission applications. A decoding circuit was designed to handle many different codes while being easily configurable.

The decoder was designed using CAD software and implemented in CMOS ASIC technology. It was used in conjunction with an encoder previously designed and fabricated to experimentally prove the error control performance of a BCH(15,5) based runlength limited code.

The ASIC was found to perform according to its specifications and the error control power of runlength limited error control codes was practically verified.

Contents

List of Figures

Chapter 1

Introduction

In recent years, digital data communication and storage systems have become increasingly more important because of their many advantages. From large-scale high-speed data networks for the exchange, processing and storage of data to the common compact disc, the digital approach has provided a whole set of new achievements in terms of features and quality. Another good example is the telephone system, the digitalisation of which is offering several new services to the general public.

In addition to filters to minimise the effects of noise, digital systems make use of appropriate coding techniques to increase reliability. By proving in 1948 that proper encoding of information can reduce the errors induced by a noisy channel to any desired level, Claude Shannon motivated extensive study and research into the development of efficient coding schemes.

Channel coding has been divided into two different subjects, line and error control coding, which have been studied separately. Real systems often make use of both approaches and, for that reason, there has been an effort to combine the two techniques into a single code [1, 2].

This dissertation is centred around the implementation of a decoding device for a novel class of codes which combine error control with certain line coding characteristics. Chapter 2 introduces some of the theory behind the project by considering line and error control coding separately and then presenting a class of runlength limited error control codes together with its mathematical description.

Chapter 3 is concerned with architectures for encoding and decoding of such codes. After introducing a subclass of error control codes important for being particularly suitable for implementation, it presents architectures for encoding and decoding of both standard and runlength limited error control codes.

The practical implementation of a flexible runlength limited error control decoder in ASIC technology is detailed in chapter 4, from hardware design to simulation and device testing. The CAD tools utilised in the design are also presented.

Chapter 5 describes an experiment which proves the error control performance of runlength limited error control codes. The dissertation concludes with an overview of the project, a brief discussion of the results and some ideas for future work.

List of Tables

Chapter 2

Channel coding

2.1 Introduction

An ideal communications channel would be absolutely transparent to the data flowing through it. For the digital case, this can be understood as a channel 100% error free. Such a channel would always respond to any particular symbol on its input with that same symbol on the output, no matter what sequence of digits had been sent before.

Real channels are often far from this ideal situation. A transmitted digit can be received as a different one for a variety of reasons, which include noise, intersymbol interference (ISI) and timing errors. The purpose of channel coding is to adapt the source signal to a particular channel (optical fiber, magnetic tape, coaxial cable), so that reliability is improved.

Channel coding is divided into two main areas, line and error-control coding, each of which has traditionally been studied separately. The next section will briefly consider line coding and some of the ideas behind it. Section 2.3 is about error-control coding, with special focus on the class of linear block codes.

Finally, a novel class of codes [1] which combine error-control and certain line coding characteristics will be presented in section 2.4. The theoretical description of such codes will be introduced.

2.2 Line coding

Line coding is the conversion of source data to a format such that detection in the presence of the usual transmission constraints is facilitated. A line code is meant to:

- minimise the effects of ISI and noise so that a sufficiently low transmission error rate is achieved;
- enable the extraction of a timing reference from the data stream.

The former item is closely related to the type of channel in use. If a channel is for base-band transmission and has low frequency cut-off, then the coded data spectrum should not exhibit important low frequency components, DC in particular. On the other hand, extraction of a timing reference is usually made possible simply by assuring a reasonable density of transitions in the data stream. Clock extraction can then be performed by using a phase-locked loop with the zero crossings as reference.

2.2.1 Runlength and disparity

Let us consider the case of binary data being transmitted over a channel and suppose that no line coding is used. If data consists of a random sequence where each symbol is equally likely to occur, and if any n consecutive bits are unrelated, long sequences of just '0' or just '1' become less and less likely. It is reasonable to expect that at the far end of the channel the receiver will be presented with some transitions, which can be used to regenerate the clock. Nevertheless, no bound exists for the number of consecutive like symbols (or runlength) that can be sent through the channel.

Also, the total number of ones transmitted during a certain period can be significantly different from the corresponding number of zeros, which means that the data spectrum can exhibit a DC component. It has been proved that such component can be eliminated by bounding the running digital sum (or running disparity[1]) of the channel data.

Both disparity and runlength can be limited by the controlled introduction of redundancy into the original data. One approach is to increase the transmission bit rate and

[1]This is a measure of the imbalance between the number of transmitted zeros and ones.

insert extra bits, as in the case of the 3B4B line code. Here, for each block of 3 bits, a binary code word of length 4 is transmitted. Limited runlength is achieved by excluding the all-0 and all-1 code words and choosing those with the greater number of transitions. On the other hand, if a DC null is desired, code words with zero disparity will be preferred. Eventually, some message words will correspond to two different code words of opposite disparity, which shall be used alternately to keep the running disparity bounded.

Another approach consists of maintaining (or even decreasing) the transmission rate and using multi-level instead of binary code words. Again, redundancy is introduced since there are fewer message words than possible code words. The previous concepts of runlength and disparity are still applicable.

2.3 Error-control coding

Error-control coding consists of introducing redundancy in the data stream so that eventual transmission errors can be detected or even corrected. The simplest (but not the most efficient) way of achieving protection against errors consists of a repetition code, where each bit of data is transmitted a fixed number of times. The receiver checks for agreement between corresponding received bits and decides that no errors occurred only if all of them are the same. Error correction can then be achieved by taking a 'majority vote' to decide which symbol is most likely to have been transmitted or simply by requesting the retransmission of the corrupted data.

Error-control systems can be of two types:

Automatic repeat request (ARQ) The receiver is able to detect the occurrence of errors but cannot correct them. If errors are detected, retransmission of the corrupted data is automatically requested. This method requires a return path from the receiver to the source, which limits its application. It is also not viable if the closed loop transfer time is long or the channel is very noisy. In the latter case, a particular block of data could have to be transmitted several times before a 'clean' version was received, thus significantly reducing the effective transmission rate.

Forward error correction (FEC) The receiver is able not only to detect errors but also

to correct them. Coding schemes are usually more complex than in the previous case but no return path is needed. Noisier channels require more powerful codes but the principle is still applicable.

Systems may combine the two approaches.

2.3.1 Error control code categories

ECCs can be separated into two distinct categories, block and tree codes. In the former case, message bits are grouped to form message words (or vectors[2]). Encoded data consists of code words which depend on the current message word only. Tree codes constitute a different approach, where encoded information depends both on the current message information and the present state of the encoder, which has memory. This means that the same piece of information can in fact assume different forms after encoding.

Codes can also be split into linear and non-linear codes. Linear codes are well structured and possess many algebraic properties which make them more attractive to study and use. However, they may be non-optimal in many senses.

Convolutional codes are an important subset of tree codes, which are both linear and time invariant. Examples of linear block codes are the standard Hamming and BCH codes (for random-error control) and the Fire codes (for burst-error control).

Error control codes may be binary or multi-level. However, only binary linear block codes will be examined in the following subsections, since they constitute an important part of the theory behind this study.

2.3.2 Linear block codes

Consider the situation where data is available in words of length 4, each of them with non-zero probability. If data is transmitted 'as is' and errors occur, the receiver will not be able to detect such errors. Altering one or more bits in any word will always simply change it into another valid message word.

[2]In what follows, the terms 'word' and 'vector' will be used interchangeably to designate a binary n-tuple.

Original data	Parity checking	Hamming(7,4)
0000	0 0000	000 0000
0001	1 0001	011 0001
0010	1 0010	110 0010
0011	0 0011	101 0011
0100	1 0100	111 0100
0101	0 0101	100 0101
0110	0 0110	001 0110
0111	1 0111	010 0111
1000	1 1000	101 1000
1001	0 1001	110 1001
1010	0 1010	011 1010
1011	1 1011	000 1011
1100	0 1100	010 1100
1101	1 1101	001 1101
1110	1 1110	100 1110
1111	0 1111	111 1111

Table 2.1: Error-control coding examples

Also consider a second example where an extra bit has been added to each message word, so that the total number of ones (or *parity*) in each code word is even (see table 2.1). Now, if a single error occurs, the received code word will be invalid and the receiver will know that something happened, though it will not know exactly what. A code like this is called a *parity checking* code and is able to detect the occurrence of any odd number of errors.

A more elaborate example is the Hamming (7,4) code. Starting from the same words of length 4, three redundant bits are added. Now, if a single error occurs, the received vector will still be closer to the original code vector than to any other vector in the list, and such an error can be corrected. The concept of vector weight will be used to define distance between two code vectors:

Definition 2.1 *The weight $w(\mathbf{v})$ of a vector $\mathbf{v}$ of binary elements is the number of '1's that are present in it.*

Definition 2.2 (Hamming distance) *The Hamming distance d between two vectors of binary elements with the same length is the number of components in which they differ, i.e.,*

$$\forall \mathbf{v}, \mathbf{w} \in V_n \quad d(\mathbf{v}, \mathbf{w}) = w(\mathbf{v} + \mathbf{w})$$

where V_n is the set of all binary n-tuples and the plus sign stands for modulo-2 addition.

The number of random-errors that can be detected and/or corrected by using a certain code depends on its *minimum distance*, which is defined as follows:

Definition 2.3 (Minimum distance) *The minimum distance $d_{\min}$ of a block code C is the least of the distances between any two distinct code words in that code, i.e.,*

$$d_{\min} = \min\{d(\mathbf{v}, \mathbf{w}) : \mathbf{v}, \mathbf{w} \in C, \mathbf{v} \neq \mathbf{w}\}.$$

A block code with minimum distance $d_{\min}$ is capable of detecting *all* error patterns of $d_{\min} - 1$ or fewer errors, but not all error patterns of $d_{\min}$ or more errors. For that reason, $d_{\min} - 1$ is said to be the *random-error-detecting capability* of such code. For example, the parity checking code is single-error-detecting ($d_{\min} = 2$) whereas the Hamming(7,4) code is double-error-detecting ($d_{\min} = 3$).

Similarly, a block code which is capable of correcting all error patterns of t or less errors, but not all error patterns of $t + 1$ or more errors, is said to have *random-error-correcting capability* t. The random-error-correcting capability of a block code C is related to its minimum distance by the equality

$$t = \left\lfloor \frac{d_{\min} - 1}{2} \right\rfloor,$$

where $\lfloor x \rfloor$ denotes the greatest integer which is not greater than x. The Hamming(7,4) code is an example of a single-error-correcting code.

Closer inspection of each of the codes described earlier in this subsection would show that in both cases the sum of any two code vectors is also a code vector. In fact, both codes are linear codes as they conform to the following definition [3]:

Definition 2.4 (Linearity) *A binary block code of length n and with 2^k code words is called a* linear (n, k) *code if and only if its 2^k code words form a k-dimensional subspace of the vector space of all the n-tuples over the field $GF(2)$.*[3]

Linear codes have a very well defined structure, which can be used to reduce the coding complexity to a great extent. Also, they can be described systematically in terms of matrix representation. Another consequence of the definition is the presence of the null vector in all linear codes. In fact, adding a vector to itself will result in the null vector, which must therefore be a valid code vector.

2.3.3 Matrix representation

Since an (n, k) linear code C is a vector subspace of dimension k, it is possible to find k linearly independent code words $\mathbf{g}_{k-1}, \mathbf{g}_{k-2}, \ldots, \mathbf{g}_0$ in C such that every code word $\mathbf{v}$ in C is a linear combination of such code words, i.e.,

$$\mathbf{v} = \sum_{i=1}^{k} u_{k-i}\, \mathbf{g}_{k-i}, \tag{2.1}$$

where $u_i \in \{0, 1\}$ for $0 \leq i \leq k-1$. Equation (2.1) can be written in matrix form, as follows:

$$\begin{aligned} \mathbf{v} &= \mathbf{u} \cdot \mathbf{G} \\ (v_{n-1}, v_{n-2}, \ldots, v_0) &= (u_{k-1}, u_{k-2}, \ldots, u_0) \cdot \begin{bmatrix} \mathbf{g}_{k-1} \\ \mathbf{g}_{k-2} \\ \vdots \\ \mathbf{g}_0 \end{bmatrix} \end{aligned} \tag{2.2}$$

Matrix $\mathbf{G}$ generates the (n, k) code and is called a *generator matrix* for C. Any set of k linearly independent code words can be used to form a generator matrix for an (n, k) linear code, which is completely defined by such a matrix. In fact, for each message word $\mathbf{u}$ the corresponding code word $\mathbf{v}$ can be found by computing $\mathbf{u} \cdot \mathbf{G}$.

[3] This is the Galois field $(\{0, 1\}, +, \cdot)$, where addition is modulo-2 (logic EXOR) and multiplication is the logic AND operation.

The generator matrix for the Hamming(7,4) code in table 2.1 (page 15) is the following:

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

For this matrix, equation (2.2) leads to

$$\begin{aligned} v_6 &= u_3 + u_2 + u_1 = b_2 \\ v_5 &= u_2 + u_1 + u_0 = b_1 \\ v_4 &= u_3 + u_2 + u_0 = b_0 \\ v_3 &= u_3 \\ v_2 &= u_2 \\ v_1 &= u_1 \\ v_0 &= u_0 \end{aligned} \tag{2.3}$$

Now it is clear that the redundant bits b_i, $i = 0, 1, 2$, in the Hamming(7,4) code are also parity bits. However, each of them depends on some of the message bits only. The other bits in the code word are simply the message bits. A code in this form is said to be in *systematic form*, as code words are divided into two parts, parity ($n - k$ bits) and message (the unaltered k message bits). Generator matrices of codes in systematic form can be written as $\mathbf{G} = [\mathbf{P}\ \mathbf{I}_k]$, where $\mathbf{I}_k$ denotes the $k \times k$ identity matrix.

Another useful matrix which is associated to every linear block code is the parity-check matrix $\mathbf{H}$. It can be derived from the code's generator matrix and constitutes an alternative description of the code. The parity-check matrix is such that

$$\mathbf{v} \cdot \mathbf{H}^T = \mathbf{0}. \tag{2.4}$$

The parity-check matrix for the Hamming(7,4) example can be obtained from (2.3) by eliminating the terms in u from the first three equations. Thus,

$$\begin{cases} v_6 + v_3 + v_2 + v_1 = 0 \\ v_5 + v_2 + v_1 + v_0 = 0 \\ v_4 + v_3 + v_2 + v_0 = 0 \end{cases} \iff \mathbf{v} \cdot \begin{bmatrix} \mathbf{I}_{n-k} \\ \mathbf{P} \end{bmatrix} = \mathbf{0}$$

$$\iff \mathbf{v} \cdot \mathbf{H}^T = \mathbf{0}.$$

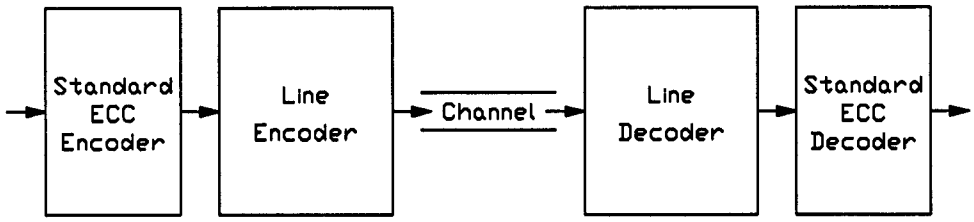


Figure 2.1: Traditional coding scheme

Equation (2.4) provides a way of detecting the occurrence of errors. If a received word $\mathbf{r}$ is not a valid code word, the product $\mathbf{v} \cdot \mathbf{H}^T$ will not be zero. The vector $\mathbf{s} = \mathbf{v} \cdot \mathbf{H}^T$ is called a *syndrome*, as it gives information not only about error occurrence but also, to a certain extent, about the error patterns themselves.

The total number of different syndromes determines the number of patterns that can be corrected. For the case of the Hamming(7,4), a total of $2^{n-k} - 1 = 7$ non-zero syndromes exists, so it is possible to correct seven error patterns. Because all the seven single-error patterns correspond to different syndromes, it is in fact possible to correct all single errors with this method, which is known as *syndrome decoding.*

2.4 Runlength limited error-control codes

In general, standard linear ECCs do not have good line coding properties. In conventional systems this problem has been overcome by using two independent coding schemes to perform error-control and line coding (figure 2.1). With such a system, the error rate performance of the error control code in use usually deteriorates since the line decoder can propagate errors. Also, for a given information rate, the overall transmission rate is increased by two different operations that both involve adding redundancy.

For the particular case of runlength limitation, a new class of codes has been proposed [1]. These are derived from conventional linear transparent[4] error control codes (LTECCs) by addition of a suitable fixed modification vector. The code obtained combines the full error control power of its parent code with the limited runlength feature, without introduction of any further redundancy [4].

[4]A code is said to be transparent if and only if the inverse of a valid code word is itself a valid code word.

Runlength limited error-control codes have applications where timing is important and a DC component in the data spectrum can be tolerated. Magnetic and optical recording, as well as several sorts of data transmission are examples of such areas.

2.4.1 Theory

Consider table 2.2. In the first column, the Hamming(7,4) code presented previously in table 2.1 is given. Because the all-0 and the all-1 code words are present, no bound exists on the number of consecutive like symbols that may be transmitted. In the second column, the vector 100 0001 (it is called the *modification vector*) was added to each of the vectors in column one, forming what is called a *coset* of the original code. Column two does not contain either the all-0 or the all-1 code words, therefore it would provide limited runlength. If a different modification vector is chosen (as in column three), a different set of code words is obtained, eventually providing a different maximum runlength. Note that the codes in each of the three columns have equal distances and therefore must be equivalent in terms of error-control performance.

This example poses the problem of determining what is the minimum worst-case runlength that can be achieved for a particular code, and which modification vector provides that minimum runlength. Systematic procedures to find the optimal solution to these questions are discussed in the next two subsections.

2.4.2 Determining the minimum worst-case runlength

For a given transparent set of code words which have a one to one mapping with source words, the maximum runlength $RL_{\max}$ can be derived as follows:

$$RL_{\max} = \max\{R_{\text{end}} + R_{\text{start}}, R_{\text{mid}}\}$$

where R_{end} and R_{start} are respectively the maximum numbers of consecutive zeros or ones at the end and at the start of any code word, and R_{mid} is the maximum number of consecutive zeros or ones in any code word.

Given an (n, k) LTECC with generator matrix G, the minimum attainable values for R_{start}, R_{mid} and R_{end} (which will be denoted by $\min R_{\text{start}}$, $\min R_{\text{mid}}$ and $\min R_{\text{end}}$) for any

Hamming(7,4)	Modified 100 0001	Modified 000 0011
000 0000	100 0001	000 0011
011 0001	111 0000	011 0010
110 0010	010 0011	110 0001
101 0011	001 0010	101 0000
111 0100	011 0101	111 0111
100 0101	000 0100	100 0110
001 0110	101 0111	001 0101
010 0111	110 0110	010 0100
101 1000	001 1001	101 1011
110 1001	010 1000	110 1010
011 1010	111 1011	011 1001
000 1011	100 1010	000 1000
010 1100	110 1101	010 1111
001 1101	101 1100	001 1110
100 1110	000 1111	100 1101
111 1111	011 1110	111 1100
$RL = \infty$	$RL = 8$	$RL = 9$
$d_{\min} = 3$		

Table 2.2: Runlength limitation examples

coset of the ECC can be obtained directly from **G**, in the following way [1]:

$$\min R_{\text{start}} = \text{the number of consecutive linearly independent (LI) columns at the start of } \mathbf{G} \quad (2.5)$$

$$\min R_{\text{mid}} = \text{the number of consecutive LI columns in } \mathbf{G} \quad (2.6)$$

$$\min R_{\text{end}} = \text{the number of consecutive LI columns at the end of } \mathbf{G} \quad (2.7)$$

It is a consequence of these results that the ordering of the columns of **G** can significantly affect the minimum runlength achievable ($\min RL_{\max}$) for each particular code. However, since only the subclass of *cyclic codes* will be considered experimentally (chapter 3), this fact will not be emphasised.

2.4.3 Finding a suitable modification vector

Having determined $\min RL_{\max}$ for a particular code, the next step consists of determining a modification vector which achieves this minimum runlength. The parity check matrix **H** is used for that purpose.

By performing row operations, **H** can be rearranged to obtain a new parity check matrix, $\tilde{\mathbf{H}}$, which should have [1]:

1. at least one row with the last $n - (\min R_{\text{start}} + 1)$ elements all zero;
2. at least one row with the first i and the last $n - (\min RL_{\max} + 1) - i$ elements all zero, for $i = 1, 2, \ldots, n - (\min RL_{\max} + 2)$;
3. at least one row with the first $n - (\min R_{\text{end}} + 1)$ elements all zero.

The rows of $\tilde{\mathbf{H}}$ which satisfy these conditions are then used to form matrix **S**. The solutions of the equation

$$\mathbf{m} \cdot \mathbf{S}^T = \mathbf{1} \quad (2.8)$$

constitute suitable modification vectors.

If $\min R_{\text{mid}}$ was used instead of $\min RL_{\max}$ in condition 2, not only the runlength bound would be satisfied but also an eventually tighter bound might be reached for the "inner runlength" of each code word, which would increase the number of transitions.

2.4.4 Illustrative example

The process of limiting the runlength of a linear transparent ECC will now be illustrated by means of an example. Both the runlength bound and suitable modification vectors are calculated for a cyclic version of the BCH(15,5) code. The results obtained are of use in chapter 5.

Consider the BCH(15,5) code with generator matrix

$$\mathbf{G} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

From (2.5) to (2.7), it follows that

$$\min R_{\mathbf{start}} = \min R_{\mathbf{mid}} = \min R_{\mathbf{end}} = 5,$$

which means that the minimum runlength that can be achieved by addition of a modification vector, for this particular version of the BCH(15,5) code, is

$$\min RL_{\mathbf{max}} = \max\{5+5, 5\} = 10.$$

Modification vectors that satisfy this bound, if they exist, can be calculated by forming the parity check matrix

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

and rearranging it by performing row operations. The conditions for the rearranged matrix $\tilde{\mathbf{H}}$ are:

1. one row with the last 9 elements zero;
2. one row with the first 1 and the last 3 elements zero;
3. one row with the first 2 and the last 2 elements zero;
4. one row with the first 3 and the last 1 element zero;
5. one row with the first 9 elements zero.

So,

$$\tilde{\mathbf{H}} = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \begin{matrix} (1+2+4+6) \\ (2+10) \\ (3) \\ (4) \\ \\ \\ \\ \\ \\ (10) \end{matrix}$$

Matrix $\mathbf{S}$ is formed from the first four rows and the last row of $\tilde{\mathbf{H}}$, to give

$$\mathbf{S} = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}.$$

Vectors such as

$$\begin{aligned} \mathbf{m}_1 &= (0,1,0,0,0,0,0,0,0,0,0,0,1,0,0) \\ \mathbf{m}_2 &= (1,0,0,0,0,0,0,0,0,0,0,0,1,0,0,1) \\ \mathbf{m}_3 &= (1,0,0,0,1,0,1,0,1,0,0,1,0,0,1) \end{aligned}$$

and some others verify equation (2.8) and therefore will create codes with a maximum runlength of 10.

2.5 Summary

In this chapter, various aspects of channel coding were considered. After briefly discussing line coding, error control coding was looked at more deeply. Special attention was paid to the class of linear block error-control codes, which was described mathematically.

In the last section, a novel class of runlength limited error control codes was introduced. The theoretical description of such codes was given and illustrated by means of an example.

Chapter 3

Architectures for Encoding and Decoding of Runlength Limited Error-Control Codes

3.1 Introduction

The class of runlength limited error control codes (RLECCs) presented in the last chapter will now be considered from an implementation point of view. However, since RLECCs are derived from standard linear error control codes by an addition operation, the problem of encoding and decoding standard ECCs will be considered first.

The subclass of *cyclic codes*, which is particularly suitable for hardware implementation, will be described mathematically in the next section. Then, standard architectures for encoding and decoding of cyclic codes will be discussed in section 3.3. Finally, in section 3.4 such architectures will be modified to accommodate encoding and decoding of RLECCs.

3.2 Cyclic codes

Cyclic codes are an important subclass of linear codes. Their rich algebraic structure makes them very attractive to study and implement. In particular, cyclic codes can be implemented by using shift registers with feedback connections, resulting in simple hardware

coding solutions.

Cyclic codes are defined as follows:

Definition 3.1 (Cyclic code) *An (n,k) linear block code is called* cyclic *if and only if every cyclic shift $\mathbf{v}^{(i)}$ of a code vector $\mathbf{v}$ in C is also a code vector in C, i.e.,*

$$\forall (v_{n-1}, v_{n-2}, \ldots, v_0) \in C, \ \forall i \in \{0, 1, \ldots, n-1\},$$
$$\mathbf{v}^{(i)} = (v_{n-i-1}, v_{n-i-2}, \ldots, v_1, v_0, v_{n-1}, \ldots, v_{n-i}) \in C$$

The cyclic character provides a code with many algebraic properties. To develop them, the use of an alternative notation to the n-tuple is appropriate.

3.2.1 Polynomial representation

Cyclic code vectors can also be represented by polynomials where each component of a particular vector is treated as a coefficient, as follows:

$$\mathbf{v}(X) = v_{n-1}X^{n-1} + v_{n-2}X^{n-2} + \cdots + v_1 X + v_0.$$

Note that there is a one-to-one correspondence between the code vector $\mathbf{v}$ and the polynomial $\mathbf{v}(X)$. For example, a cyclic shift of $\mathbf{v}(X)$ can be written

$$\mathbf{v}^{(1)}(X) = X\mathbf{v}(X) + v_{n-1}(X^n + 1).$$

In general,

$$\begin{aligned} \mathbf{v}^{(i)}(X) &= X^i\mathbf{v}(X) + \mathbf{q}(X)(X^n + 1) \\ &\Updownarrow \\ X^i\mathbf{v}(X) &= \mathbf{q}(X)(X^n + 1) + \mathbf{v}^{(i)}(X), \quad \forall i \in \{0, 1, \ldots, n-1\} \end{aligned}$$

where $\mathbf{q}(X) = v_{n-1}X^{i-1} + v_{n-2}X^{i-2} + \cdots + v_{n-i}$. This means that the code polynomial $\mathbf{v}^{(i)}(X)$ is simply the remainder of the division of $X^i\mathbf{v}(X)$ by $X^n + 1$.

Many other properties of the cyclic codes can be formalised in terms of polynomial representation. Some of the most important follow [3]:

1. The non-zero code polynomial $\mathbf{g}(X)$ of minimum degree in a cyclic code C is unique.

2. $\mathbf{g}(X)$ has degree $q = n - k$.

3. $\mathbf{g}(X)$ is a factor of $X^n + 1$.

4. A binary polynomial $\mathbf{v}(X)$ is a code polynomial in C if and only if $\mathbf{v}(X)$ is a multiple of $\mathbf{g}(X)$, i.e.,

$$\begin{aligned} \mathbf{v}(X) &= \mathbf{u}(X)\mathbf{g}(X) \\ &= (u_{k-1}X^{k-1} + u_{k-2}X^{k-2} + \cdots + u_0)\mathbf{g}(X). \end{aligned} \tag{3.1}$$

Expression (3.1) suggests a method for the encoding of cyclic codes. If $\mathbf{u}(X)$ represents the information vector to be encoded, $\mathbf{v}(X)$ is the corresponding code polynomial. The polynomial $\mathbf{g}(X)$ is called the *generator polynomial* of the code, as any code vector can be derived from it.

3.2.2 Encoding

It has been shown that code vectors can be obtained by multiplying the corresponding message polynomials by $\mathbf{g}(X)$. However, the resulting code is not in systematic form.

By multiplying message polynomials by X^{n-k} and dividing the results by $\mathbf{g}(X)$, code polynomials can be generated in systematic form. For a particular message $\mathbf{u}(X)$ this means that

$$\begin{aligned} X^{n-k}\mathbf{u}(X) &= \mathbf{a}(X)\mathbf{g}(X) + \mathbf{b}(X) \\ &\Updownarrow \\ X^{n-k}\mathbf{u}(X) + \mathbf{b}(X) &= \mathbf{a}(X)\mathbf{g}(X) = \mathbf{v}(X), \end{aligned} \tag{3.2}$$

where $\mathbf{b}(X)$ is a polynomial of degree $n - k - 1$ or less. Thus, the polynomial obtained

$$\begin{aligned} \mathbf{v}(X) = &\underbrace{u_{k-1}X^{n-1} + u_{k-2}X^{n-2} + \cdots + u_0X^{n-k}}_{\text{message}} + \\ &+ \underbrace{b_{n-k-1}X^{n-k-1} + b_{n-k-2}X^{n-k-2} + \cdots + b_0}_{\text{check}} \end{aligned}$$

is in systematic form while being a valid code polynomial (it is a multiple of $\mathbf{g}(X)$).

The encoding problem has been reduced to performing a division by $\mathbf{g}(X)$. There are very efficient ways of carrying out such a division, using simple logic elements. This subject will be treated in section 3.3.

3.2.3 Decoding and error control

When a code is given in systematic form, the simplest decoding technique consists of discarding the check bits of the received code vector. Although not giving any control over eventual errors, the decoding process could not be simpler. This illustrates how easily information can be extracted from systematic code words.

It was mentioned earlier that valid code polynomials of a given code are always multiples of the code's generator polynomial. Therefore, if the decoder is presented with an invalid polynomial, errors must have occurred. In general,

$$\mathbf{r}(X) = \mathbf{v}(X) + \mathbf{e}(X),$$

where $\mathbf{e}(X)$ is the error polynomial. Errors can be detected by dividing the received polynomial $\mathbf{r}(X)$ by $\mathbf{g}(X)$ and looking at the remainder, which should be zero for no errors. The remainder is also called the *syndrome* and is equivalent to that computed as $\mathbf{s} = \mathbf{v} \cdot \mathbf{H}^T$ (subsection 2.3.3).

Each syndrome corresponds to a family of error patterns, called a coset. When a syndrome is computed for a particular received vector, it is possible to identify the most likely error pattern in the corresponding coset. Errors can then be corrected by adding that pattern to the received word.

Syndrome computation as described above can be used to perform *minimum distance* (or *maximum-likelihood*) decoding of random-error-correcting codes. Although being optimal in the error-performance sense, in the general case this method requires a look-up table to map syndromes onto the corresponding error patterns. Hence, it is inappropriate when a large number of error patterns are to be corrected.

To overcome this problem, other decoding methods which lead to simpler implementation solutions have been devised. However, these are only sub-optimal.

3.2.4 Error-trapping decoding

If some restrictions are imposed to the error patterns that are to be corrected, a direct correspondence between syndromes and error patterns can be achieved. Consider the case where errors are confined to the $n-k$ least significant positions of the received polynomial, i.e.,

$$\mathbf{e}(X) = e_{n-k-1}X^{n-k-1} + e_{n-k-2}X^{n-k-2} + \cdots + e_0.$$

Then, the syndrome of $\mathbf{r}(X) = \mathbf{v}(X) + \mathbf{e}(X)$ is equal to the error polynomial $\mathbf{e}(X)$, as $\mathbf{v}(X)$ is a multiple of $\mathbf{g}(X)$ and the degree of $\mathbf{e}(X)$ is less than that of $\mathbf{g}(X)$.

More generally, if errors are confined to any $n-k$ consecutive positions, including the end-around cases, the error pattern can be obtained by calculating the syndrome of an appropriate cyclic shift of $\mathbf{r}(X)$, say $\mathbf{r}^{(i)}(X)$. The value of i should be such that the new error polynomial $\mathbf{e}^{(i)}(X)$ has degree less than $n-k$. It can be proved [3] that the syndrome $\mathbf{s}_i(X)$ of $\mathbf{r}^{(i)}(X)$ can be calculated as the remainder of the division of $X^i\mathbf{r}(X)$ by $\mathbf{g}(X)$.

In practice, syndromes $\mathbf{s}_i(X)$ are successively calculated for each cyclic shift of $\mathbf{r}(X)$. When the syndrome becomes equal to a correctable error pattern, errors are considered to be *trapped* and correction is performed. This is called *error-trapping decoding* (ETD). Correctable error patterns are usually selected according to the code's characteristics, namely the *random-error-correcting* and the *burst-error-correcting* capabilities.

In the former case, a threshold gate which tests the syndrome weight against the error-correcting capability t of the code is used. Only when $w(\mathbf{s})$ becomes t or less are errors considered to be trapped. This strategy limits the number of correctable patterns in two ways:

1. If t or less errors occur but they are not confined to $n-k$ consecutive positions, including the end-around cases, errors are correctable in the minimum distance sense, but cannot be trapped.

2. Some codes are capable of correcting some error patterns of $t+1$ and more errors when minimum distance decoding is used. With the error-trapping approach none of these patterns will be corrected, even if errors are confined to some $n-k$ consecutive positions.

Thus, error-trapping decoding of random-error-correcting codes is not always equivalent to minimum distance decoding.

Burst-error-correcting codes assume that errors do not occur independently, but are more likely to occur close to one another. For this reason, they are particularly suitable for error-trapping decoding. If a code is capable of correcting all bursts of length l or less, errors are considered to be trapped when the syndrome register contains at least $n-k-l$ consecutive zeros. l is called the burst-error-correcting capability of the code.

3.3 Architectures

In the last section it was shown that both encoding and decoding of cyclic codes can be performed by carrying out a division by $\mathrm{g}(X)$. Very efficient circuits which are capable of performing such division can be constructed from simple logic elements, essentially modulo-2 adders (EXOR gates) and register elements.

3.3.1 The polynomial division circuit

From equation (3.2), the parity check polynomial $\mathrm{b}(X)$ that corresponds to a given message polynomial $\mathrm{u}(X)$ can be obtained by calculating the remainder of the polynomial division of $X^{n-k}\mathrm{u}(X)$ by the generator polynomial $\mathrm{g}(X)$.

The generator polynomial for the Hamming(7,4) code presented previously in table 2.1 is $\mathrm{g}(X) = X^3 + X + 1$. As an example, suppose that $\mathrm{u}(X) = X^3 + X^2$. The remainder of the division of $X^{n-k}\mathrm{u}(X) = X^3(X^3 + X^2) = X^6 + X^5$ by $\mathrm{g}(X)$ is

$$
\begin{array}{rl}
 & X^3 + X^2 + X \\
\hline
X^3 + X + 1 \;) & X^6 + X^5 \\
 & X^6 \quad + X^4 + X^3 \\
\hline
 & X^5 + X^4 + X^3 \\
 & X^5 \quad + X^3 + X^2 \\
\hline
 & X^4 \quad + X^2 \\
 & X^4 \quad + X^2 + X \\
\hline
 & X = \mathrm{b}(X)
\end{array}
$$

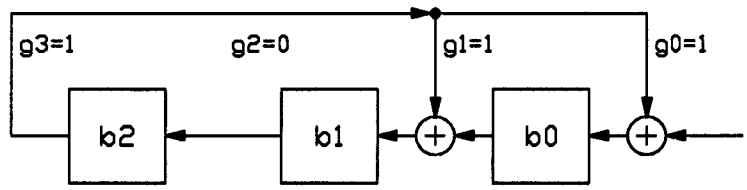


Figure 3.1: Polynomial division circuit

which confirms that the parity check bits for message $(1,1,0,0)$ are $(0,1,0)$. Note that, since the code is cyclic, code words can be formed by either appending the check bits to the original message or vice-versa.

The previous example illustrates how the remainder is calculated by repeatedly subtracting (mod-2) shifted copies of $\mathbf{g}(X)$ from the original polynomial. This is implemented by the circuit in figure 3.1, where the feed-back connections correspond to the coefficients of $\mathbf{g}(X)$. If the binary representation of $X^3\mathbf{u}(X)$ is clocked into the circuit, higher order bits first, and assuming that all the register elements are initially reset, the register contents evolve as follows:

Input	Register contents			
	0	0	0	initial state
1	0	0	1	first shift
1	0	1	1	second shift
0	1	1	0	
0	1	1	1	fourth shift
0	1	0	1	
0	0	0	1	
0	0	1	0	seventh shift

After the fourth cycle, the register contains the remainder of $\mathbf{u}(X)$ divided by $\mathbf{g}(X)$. The three extra shifts correspond to multiplying $\mathrm{u}(X)$ by X^3. This circuit is composed of $n-k=3$ register elements and requires $n=7$ shifts to calculate $\mathrm{b}(X)$.

The division of $X^{n-k}\mathbf{u}(X)$ by $\mathbf{g}(X)$ can also be performed by feeding the message bits from the other end of the register, as in figure 3.2. In this case, multiplication by X^{n-k} becomes implicit and only k cycles are needed to calculate the parity bits [3].

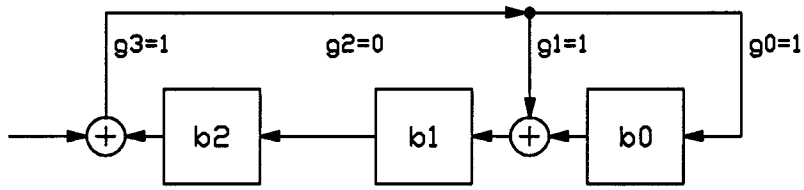


Figure 3.2: Another polynomial division circuit

If the binary representation of $\mathbf{u}(X) = X^3 + X^2$ is clocked into this second circuit, the successive register contents are

Input	Register contents			
	0	0	0	initial state
1	0	1	1	first shift
1	1	0	1	
0	0	0	1	
0	0	1	0	fourth shift

which gives the same result.

3.3.2 A polynomial encoder

Circuits in figures 3.1 and 3.2 generate the check bits for given message words and for a particular code. Figure 3.3 represents the basic architecture for a complete cyclic code encoder. It is composed of three registers, message, check and output. The feed-back connections on the check bit register depend directly on the generator polynomial of the code to be generated and form a circuit similar to that in figure 3.2.

Message bits are clocked into the polynomial division circuit and, at the same time, stored in the message input register. When a message has been fully entered the corresponding check bits become available. Code words are formed by parallel loading message and check bits into the output register. While a code word is being output (at a higher rate) check bits for the next message word are generated.

Details of the practical implementation of such a cyclic code encoder can be found in reference [5].

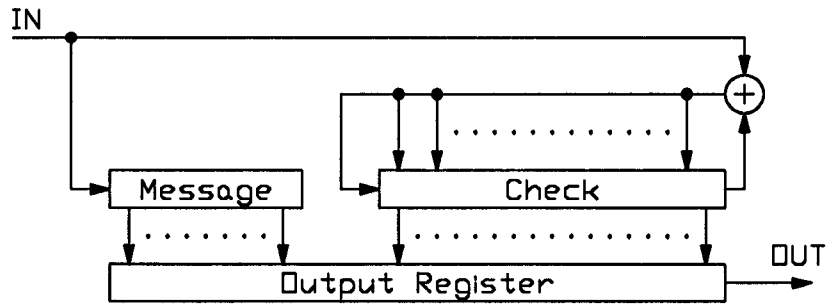


Figure 3.3: Polynomial encoder

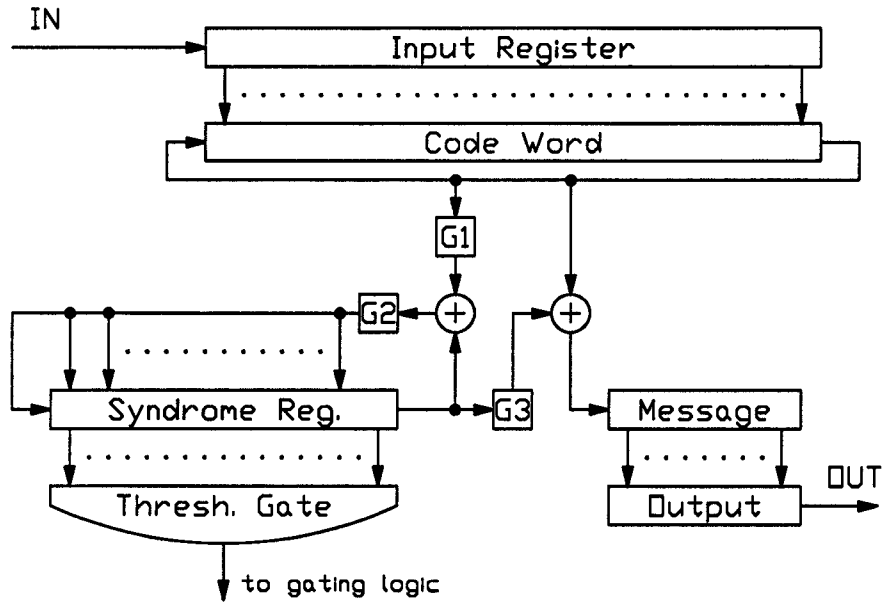


Figure 3.4: Error-trapping decoder

3.3.3 An error-trapping decoder

Since error-trapping decoding is based on the division of the received polynomial $\mathbf{r}(X)$ by the generator polynomial $\mathbf{g}(X)$ to calculate the syndrome, it is not surprising that the error-trapping decoder in figure 3.4 uses very similar building blocks to those used by the encoder.

Code word bits are clocked into the input register and parallel loaded into another register. While a new code word is being input, this register is shifted cyclically at twice the input rate, thereby clocking the code word into the syndrome register. After the first n cycles the syndrome register contains the syndrome $\mathbf{s}_{n-k}(X)$ of $\mathbf{r}^{(n-k)}(X)$. The threshold

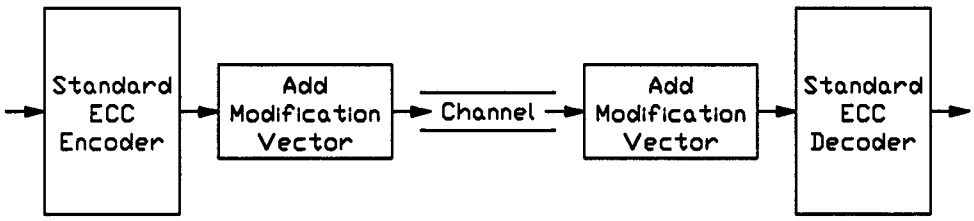


Figure 3.5: Runlength limited coding scheme

gate is then used to decide whether or not errors have been trapped by comparing the syndrome weight with the error-correcting capability of the code.

Assuming that correctable errors have occurred, if $w\left(\mathbf{r}^{(n-k)}\right) \leq t$ then errors are trapped and correspond to the $n-k$ high-order bits of $\mathbf{r}$. If not, the syndrome register has to be shifted again (a maximum of $n-1$ times), until the syndrome weight becomes less than or equal to t. Errors are corrected by serially adding the error pattern to the received code word, which is made available by shifting the code word register a second time. The decoded message is stored in the message register before it is parallel loaded into the output register, from which it is clocked out at a lower rate.

Extra logic is needed to control the resetting and loading of registers and also to decide whether gates should be open or closed. Details of the practical implementation of such an error-trapping decoder are presented in chapter 4.

3.4 Modified architectures for encoding and decoding RLECCs

Architectures for standard error control coding can be modified to accommodate runlength limitation as described in section 2.4 by inserting two extra blocks in the data path, one at each end of the communications channel (figure 3.5).

For the examples presented in the last section an extra shift register and an EXOR gate constitute all the extra logic required to obtain runlength limitation (figures 3.6 and 3.7). Therefore, from the implementation point of view, runlength limited error control coding solutions are only trivially more complex than standard error control coding.

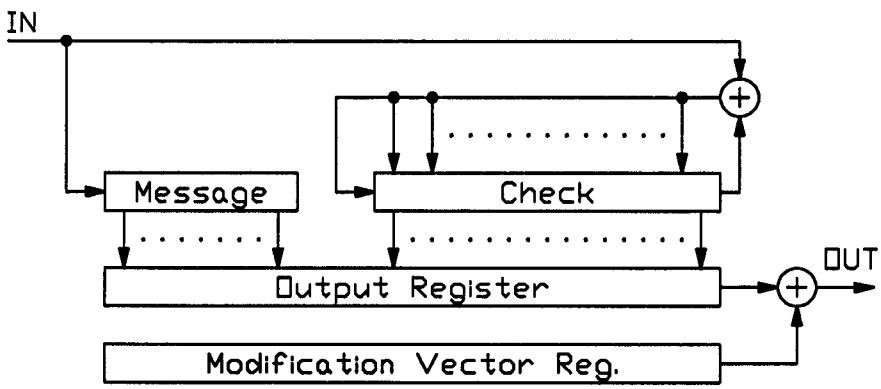


Figure 3.6: Runlength limited encoder.

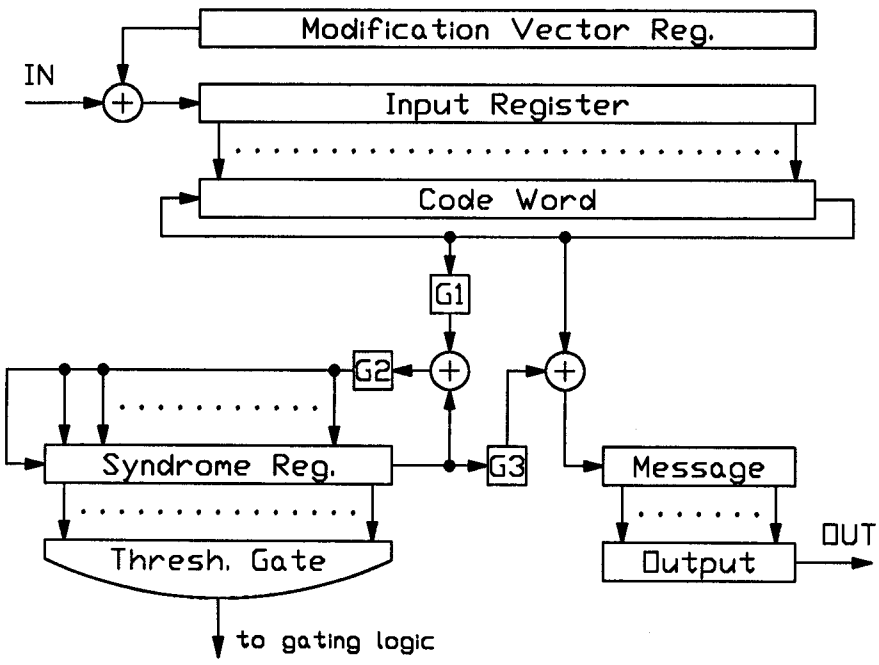


Figure 3.7: Runlength limited decoder

3.5 Summary

The subclass of cyclic error control codes particularly suitable for implementation was introduced in this chapter. Their mathematical description was used to derive efficient architectures for cyclic code encoding and decoding, which were later modified to accomodate runlength limited error-control codes.

The architecture proposed for decoding constitutes the basis for the runlength limited error control decoder the implementation of which is described in the next chapter.

Chapter 4

ASIC Implementation

4.1 Introduction

This chapter describes the ASIC implementation of a flexible runlength limited error control decoder.

Hardware design is covered in section 4.2. After presentation of the available technology, initial operational requirements are considered and a detailed description of the final circuit is given.

Section 4.3 introduces the CAD software tools that were used during the course of this project. Schematic capture and simulation facilities are presented and test strategies discussed.

Finally, in section 4.4, verification of fabricated devices is described and results are presented.

4.2 Hardware design

4.2.1 The ASIC

Digital circuits were traditionally implemented using conventional TTL or CMOS integrated circuits. Although this is adequate for simple circuits which require small amounts of logic, several problems arise when complexity increases. Printed circuit boards rapidly

become large and expensive and speed performance deteriorates as a consequence of long interconnecting lines.

ASICs (Application Specific Integrated Circuits) are a good alternative to using discrete logic, both in terms of price and performance. ASICs are compact and reliable and their power consumption is often lower than that of an equivalent discrete circuit. Moreover, ASIC manufacturers usually provide software tools that enable the user not only to design circuits but also to simulate them.

The type of ASIC chosen is manufactured by M.C.E. – Micro Circuit Engineering Ltd. using 5μm CMOS technology, and consists of an array of 1440 standard cells from which digital circuits can be constructed. This is the same ASIC used by Hutchinson [5] to implement his flexible encoder.

4.2.2 Operational requirements and specification

This device has been designed to be used in conjunction with a programmable encoder, which was already available (PECLC, see [5]). Therefore, its initial requirements were much like those of the encoder itself. Hence,

1. the device should be capable of handling any cyclic code within the following bounds:

   - maximum code word length $N_{\max} = 31$;
   - maximum message length $K_{\max} = 31$;
   - maximum number of check bits $Q_{\max} = 15$,

   providing both error detection and correction facilities;

2. the device should be able to add to the n-bit input code words an alterable modification vector of the same length, in order to recover the original error-control code words;

3. the device should provide a way of driving a parallel-in serial-out (PISO) register to facilitate the supply of a modification vector.

The first problem arose from the first requirement. In fact, although error detection can be easily implemented for the entire family of cyclic codes (e.g., by syndrome computation), the same cannot be said about error correction. The task is then to decide which error patterns to correct. The following assumptions were made:

1. the errors are confined to $q = n - k$ consecutive positions, including the end-around cases. It was shown (subsection 3.2.4) that, in this case, the error pattern can be obtained by syndrome computation;

2. The number of errors is not greater than the random-error-correcting capability (RECC, t) of the code.

These assumptions lead to the implementation of *error-trapping decoding*, as discussed in chapter 3. Because errors are considered to be trapped according to the RECC of the code, full decoding of burst-error-correcting codes may raise some problems.

Error-trapping decoding of random-error-correcting codes is not always equivalent to maximum-likelihood decoding. It is most effective for decoding single-error and some double-error-correcting codes. However, for other codes (with larger t) it becomes impossible to correct all error patterns of weight t or less, as some of them do not respect the first assumption above. This means that some (or even much) of the error-correcting capability is sacrificed.

These limitations have been accepted since the device was intended for test purposes only, and flexibility was an important requirement. Moreover, tests were to be carried out in the presence of white noise, which leads to random-errors. Finally, a burst-error-correcting decoder would require even more logic and force the design to be partitioned into two chips instead of a single one, with the available technology.

4.2.3 Design partitioning

In order to make it easier to understand and debug the circuit was modularised. Each module performs a specific task and could be tested separately. Modules can be divided into sub-modules which perform simpler operations. Figure 4.1 shows how modules relate to one another.

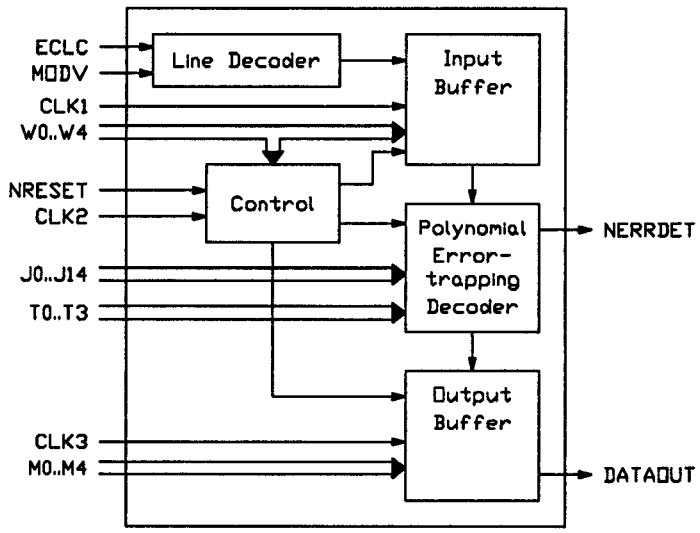


Figure 4.1: Design block diagram

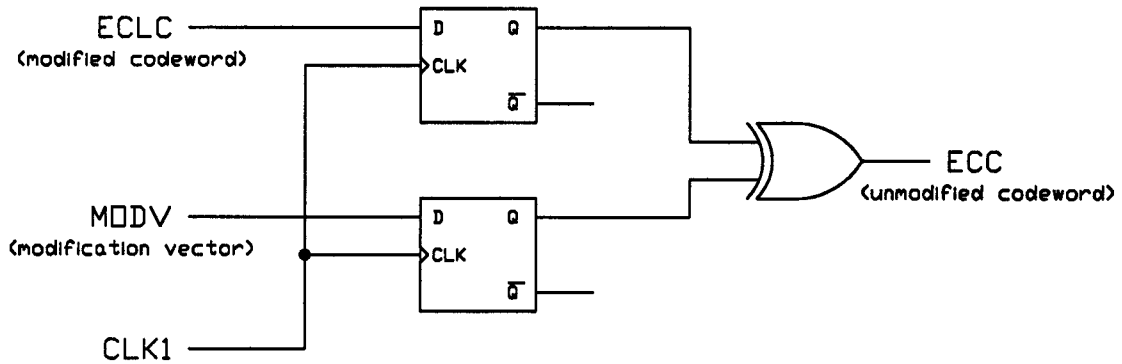


Figure 4.2: The runlength limited decoder (LNDECO)

Module 1: The runlength limited decoder

This is the simplest module in the entire circuit. Its function is to add to the incoming data an externally supplied modification vector, in order to recover the unmodified code words. An EXOR gate is used to serially perform the mod-2 addition after both signals have been latched by two flip-flops (see figure 4.2). Note that ECLC and MODV pins are exchangeable as the circuit is perfectly symmetrical.

It is very important that the modification vector is clocked into the module at the same rate as are code words. It is also crucial that the first bits of each vector are clocked in by the same clock transition. The supplying of a modification vector will be discussed in

subsection 4.2.4.

Module 2: The error-trapping decoder

Error-trapping decoding of cyclic codes was already discussed in chapter 3. Still, a few details concerning flexibility have to be pointed out. There are three important things to note:

1. the syndrome register (SYNREG) should be of variable length and further, accept different generator polynomials. As direct mapping between the generator polynomial and the feedback connections on the syndrome register exists, flexibility at this level was achieved by using AND gates to validate each connection;

2. the random-error-correcting capability varies from code to code and therefore a variable threshold gate was needed. It was implemented in two cascaded blocks, the first of which (WEIGHT) calculates the syndrome weight in binary form. Note that this calculation may be done over the full syndrome register since unused elements will always contain zeros. The second block (CMP4LE) compares the syndrome weight with the externally supplied RECC and decides whether errors are trapped or not. Both blocks are combinational;

3. decoding is to be performed in real time, i.e., one decoded message should be output per each input code word. It was therefore convenient that the decoding process took a fixed number of steps for each code, independently of where errors might occur.

Figure 4.3 presents the decoder block diagram. Consider a code of length n, with q parity check bits and random-error-correcting capability t. Then, only the q rightmost elements of SYNREG will be used. Absence of feedback connections for the remaining elements simply exclude them from the circuit.

Full decoding of each code word takes $2n$ steps. The initial conditions are that the syndrome register is reset, gates one and two are open and gate three is closed, and are achieved after one clock cycle. Then the code word (call it r) is shifted into the register, the check bits preceding the message bits, which takes n more steps. Gate one is then

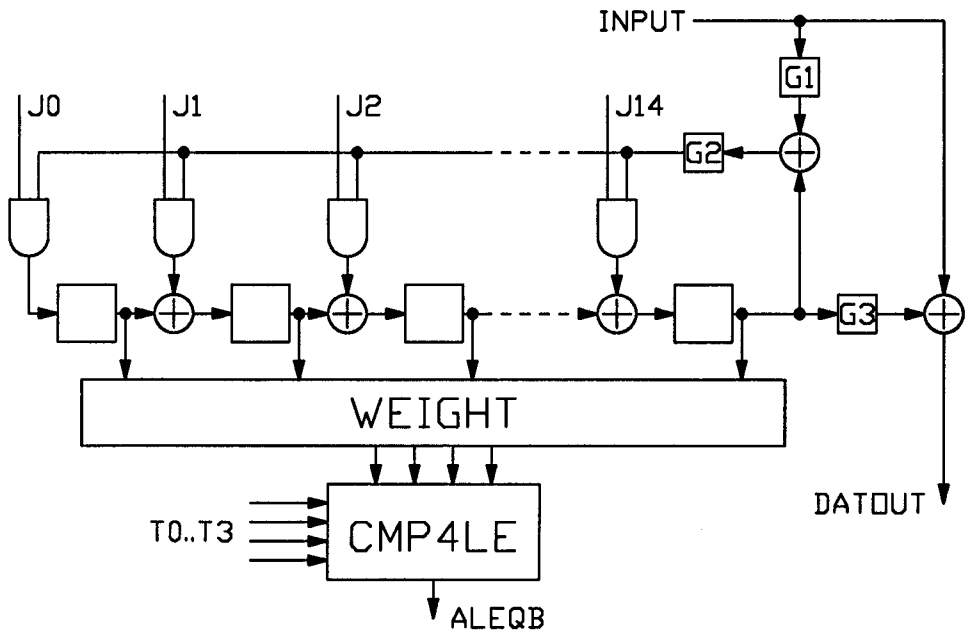


Figure 4.3: The error-trapping decoder (ERRTRP)

closed and the syndrome weight evaluated. Assuming that no more than t errors occurred and that they are confined to q consecutive positions of $\mathbf{r}$, two things can happen:

1. the syndrome weight is not greater than t. In that case, errors are considered to be trapped in the syndrome register and correspond to the q bits that were input first (the check bits). Gate two is then closed and gate three is open. The code word has to be shifted again into the decoder such that errors can be corrected. Correction is performed by serially adding the error pattern to the code word. Gate three does not need to be closed again since after q shifts the syndrome register will contain only zeros;

2. the syndrome weight is greater than t. Then errors have occurred but are not confined to the first q bits of $\mathbf{r}$. There are two possibilities:

   i) errors are confined to some other q consecutive bits, excluding the end-around case. Then the first bit is correct.

   ii) errors are confined to q consecutive bits but only considering end-around. In this case, the first bit may be in error.

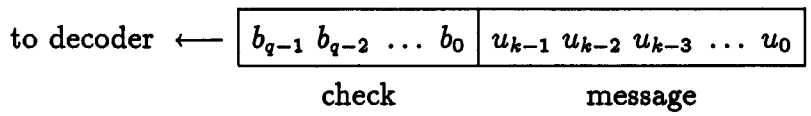


Figure 4.4: Code word format

One must not forget that the first q bits are check bits and will be discarded, so it is not important whether they are corrected or not. Assumption is made that the first bit is correct and the syndrome register is shifted again until the syndrome weight becomes less than or equal to t. When this occurs, gate two is closed and gate three is open, so that errors can be corrected as before.

It is important to note that if errors are confined to q consecutive positions they will always be trapped. The worst case is when errors can only be trapped after $n-1$ shifts. This corresponds to the errors being confined to the q bits from the n-th to the $(q-1)$-th (end-around case). When this happens, only one more step is taken to correct the message bit. The remaining errors correspond to check bits and are discarded by resetting the syndrome register again.

The output of the threshold gate may be used to obtain information about error detection and correction. If no errors occur, or if errors are not more than t and are confined to the check bits, signal ALEQB should be at logic level 1 after the first $n+1$ shifts. On the other hand, if ALEQB is at 0 after $2n$ cycles it means that errors could not be trapped, and therefore correction was not performed. This signal corresponds to the external pin NERRDET.

The present module assumes that the input vector $\mathbf{r}(X)$ may be written as

$$[\mathbf{u}(X) + X^k \mathbf{b}(X)] + \mathbf{e}(X)$$

where $\mathbf{b}(X)$ represents the parity check bits computed as the remainder of the descendent division of $X^q\mathbf{u}(X)$ by $\mathbf{g}(X)$. It is also assumed that code words are fed in from the most significant bit (see figure 4.4).

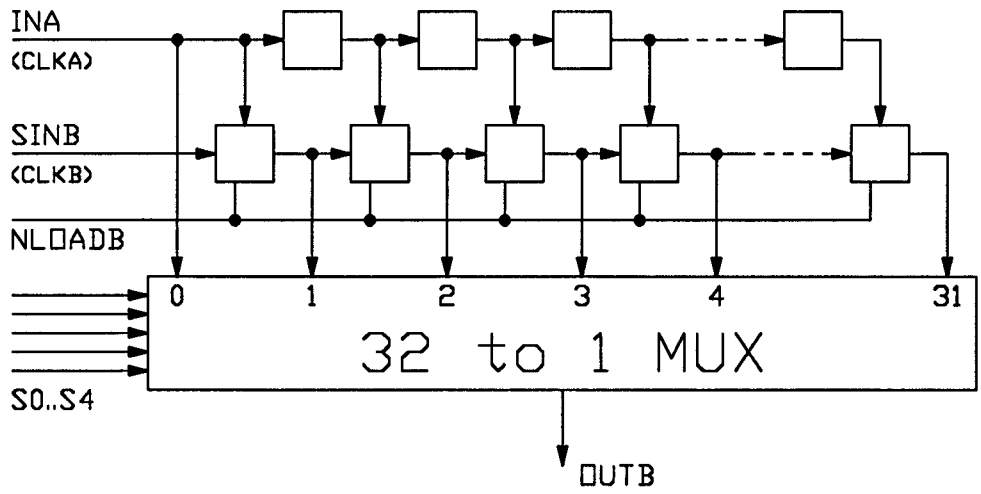


Figure 4.5: The input and output buffers (PIPELI)

Modules 3 and 4: The input and output buffers

These modules will be presented at the same time since they are actually two copies of the same block (PIPELI). Their function is to serially receive a word of variable length at an input clock rate and parallel load it into another register. It then becomes possible to process the first word at a different rate while a second word is input. The basic scheme is presented in figure 4.5.

Input data is clocked in the first register by CLKA. Meanwhile, data already present in the second register can be clocked out by CLKB, or even circulated if the serial input SINB is used. The variable length feature is achieved by using a multiplexer to choose one from the parallel outputs of register 2. If NLOADB is low when a transition in CLKB occurs, data stored in the first register will be parallel loaded into the second register. The fact that the actual input is loaded into the first element of register 2 saves one element in register 1 and, more important than that, reduces the overall delay by one cycle of CLKA.

When used as the input buffer, this module accepts code words from module 1 at the code word clock rate (CLK1) and cyclically shifts them at the decoder clock rate (CLK2). This makes it possible for each vector to be decoded to be shifted twice into module 2, as required. The effective register length is the code word length.

When used as the output buffer, this module collects the decoded messages from module 2 at the decoder rate and clocks them out (DATAOUT pin) at the message clock rate

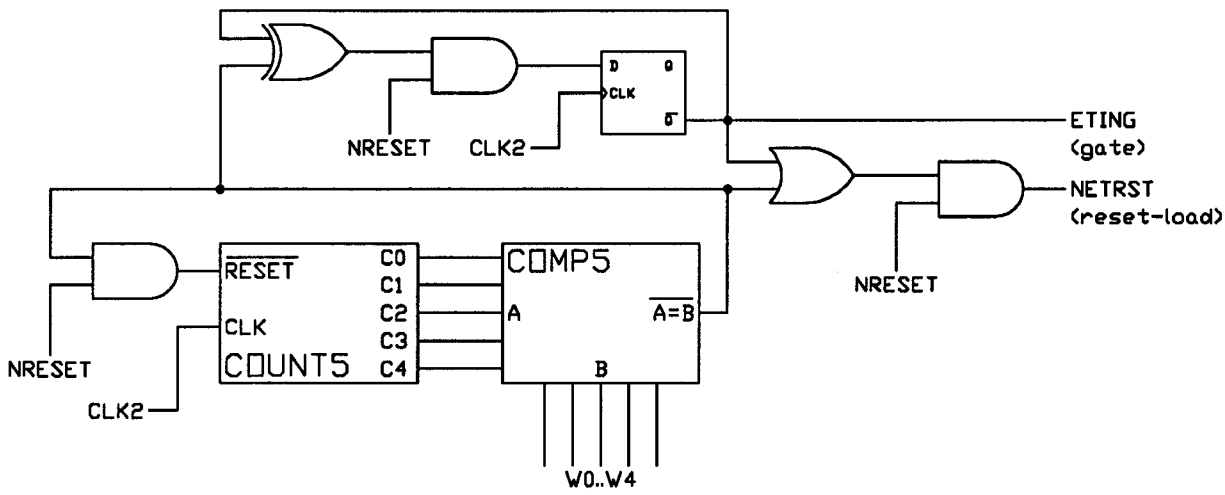


Figure 4.6: The control module (CTRLSG)

(CLK3). The effective register length should be the message word length.

This module also performs another feature. As register two has length 31 a multiplexer input was still available, so it was decided to use it to bypass the module when the register length was programmed to be zero. This feature has been utilised at the testing stage.

Module 5: Control logic

The control module (figure 4.6) is in charge of generating appropriate gating, reset and load signals required by the other modules. It receives information from pins W0 to W4 (code word length) and from pin NRESET, and runs at the decoder clock rate (CLK2). Two basic signals are generated, a gate signal and a reset-load signal. The former controls the first gate of module 2, and together with ALEQB controls gates 2 and 3 as well. The latter is interpreted by module 2 as reset and by modules 3 and 4 as load.

4.2.4 Overview

In the last subsection modules were considered separately. The circuit will now be discussed as a whole by means of an example. It will be based on the BCH(15,5) code which has the following parameters [3]:

- code word length $n = 15$;

- message length $k = 5$;
- number of check bits $q = n - k = 10$;
- random error correcting capability $t = 3$;
- generator polynomial $\mathbf{g}(X) = X^{10} + X^8 + X^5 + X^4 + X^2 + X + 1$.

Configuration pins

There are four sets of pins that are used to configure the device according to the code's parameters. The code is specified by a word called **J**, derived from $\mathbf{g}(X)$ in the following manner:

1. Take the binary representation of $\mathbf{g}(X)$, i.e., 10100110111.
2. Drop the leading "1". This gives 0100110111.
3. Add $15-q$ zeros after the rightmost bit to obtain a 15-bit word, i.e., 010011011100000.

The added zeros correspond to those elements in the syndrome register that will not be used. J pins should be set according to the word obtained, as follows:

J14	J13	J12	J11	J10	J9	J8	J7	J6	J5	J4	J3	J2	J1	J0
0	1	0	0	1	1	0	1	1	1	0	0	0	0	0

The rest of the configuration pins give information about word length, message length and error-correction capability, and should be set in the following way:

W4	W3	W2	W1	W0
0	1	1	1	1

$n = 15;$

M4	M3	M2	M1	M0
0	0	1	0	1

$k = 5;$

T3	T2	T1	T0
0	0	1	1

$t = 3.$

Clock signals

Three clock signals are required for the device to operate:

CLK1 is the input clock and its frequency corresponds to the input bit rate. Code word and modification vector bits are clocked into the decoder on its negative edges.

CLK2 is the decoder clock and its frequency should be twice that of CLK1. It is used by all circuit modules except the RL decoder (module 1).

CLK3 is the output clock. Its frequency should be related to that of CLK1 by

$$f_{\mathtt{CLK3}} = \frac{k}{n} f_{\mathtt{CLK1}}.$$

Phase relationships between clock signals are very important, especially at the beginning of each code word. There is another important constraint to be observed, the maximum speed of operation, i.e., $\max[f_{\mathtt{CLK2}}] = 1/T_{\min}$. Figure 4.7 illustrates these constraints. The last signal is the reset signal and is described next.

The reset signal

This signal is used to initialise the device. It has to be held low once until the falling edge of CLK2 to which the other clock signals relate. This edge becomes "instant 0" and from then on the control module generates internally its own reset-load signal, as discussed before.

It is also possible to apply to the NRESET pin a frame pulse of period $2nT$ since it will only be ANDed to the internal pulse. This type of signal will be more commonly available in practice and has the advantage of forcing the decoder to the initial state externally as well. It should help the decoder to recover in case it enters any unused state for some reason.

Input and output signals

There are essentially two input and two output signals:

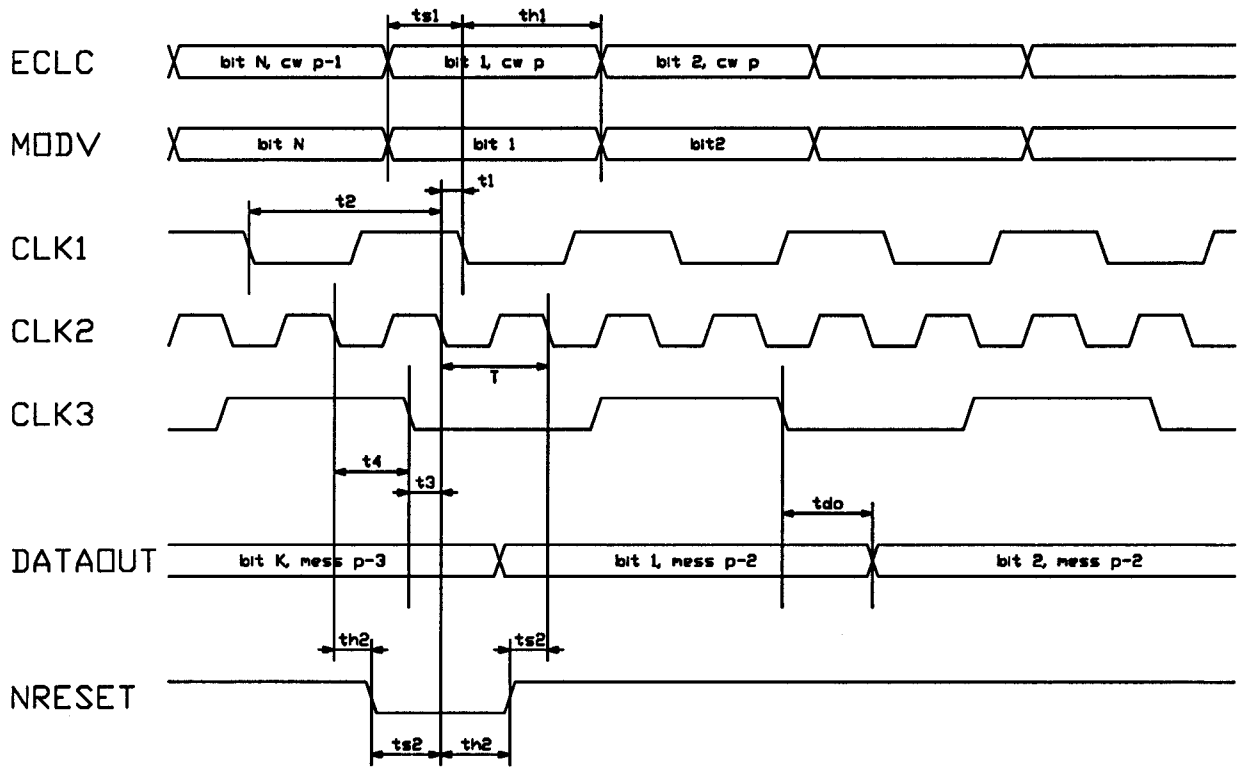


Figure 4.7: System timing (bit level)

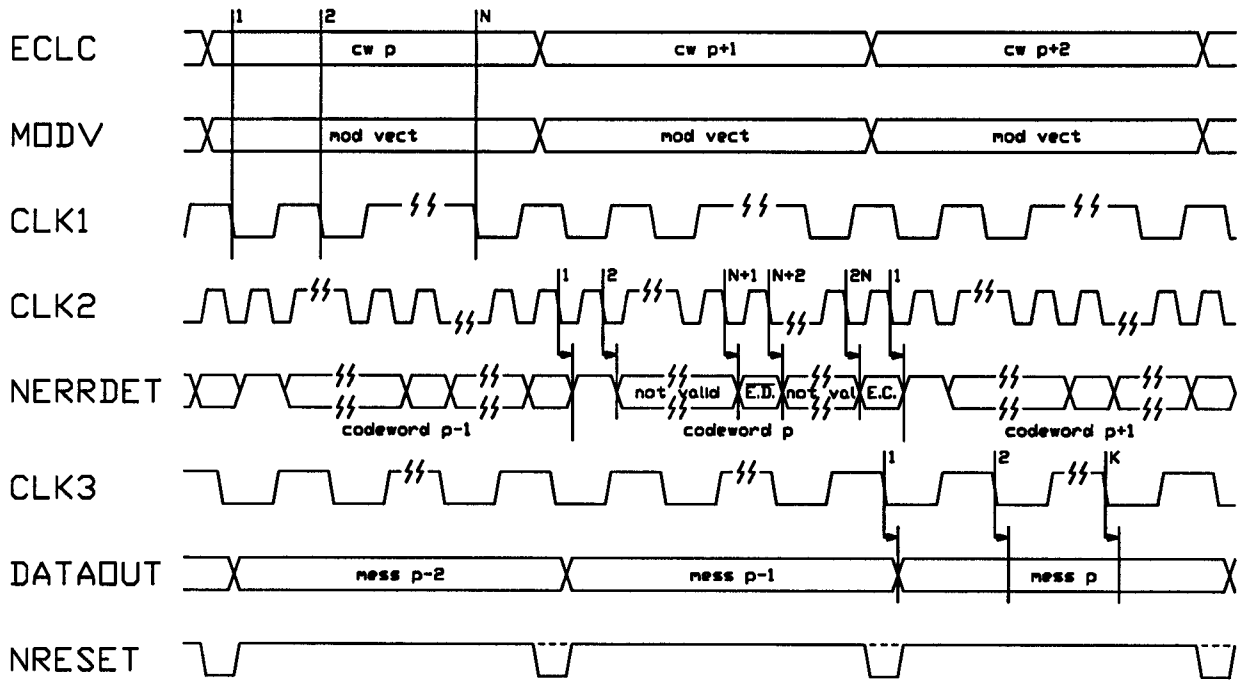
ECLC is the data input signal. It accepts modified code words to be decoded. Code word bits are clocked in the decoder on negative edges of CLK1.

MODV is the modification vector input signal. Modification vectors are supplied to module 1 serially through this pin, being clocked in on negative edges of CLK1. It is functionally equivalent to the ECLC input and this signals could be swapped.

DATAOUT is the data output signal. Decoded messages are clocked out on negative edges of CLK3. Figure 4.7 illustrates the timing for these signals.

NERRDET is the error-control output signal. It is clocked out on negative edges of CLK2 and gives information about error detection and correction, in a multiplexed form.

Although the throughput of the device is one, i.e., one message word is output per each input code word, there is a latency associated with each output signal. DATAOUT has a latency of two, i.e., the first decoded message is output only when the third code word is being input, and NERRDET has latency one. Figure 4.8 illustrates what has just been said.



Note: $\overline{\text{E.D.}}$ — errors detected in message bits (active low);
E.C. — errors corrected (active high).

Figure 4.8: System timing (frame level)

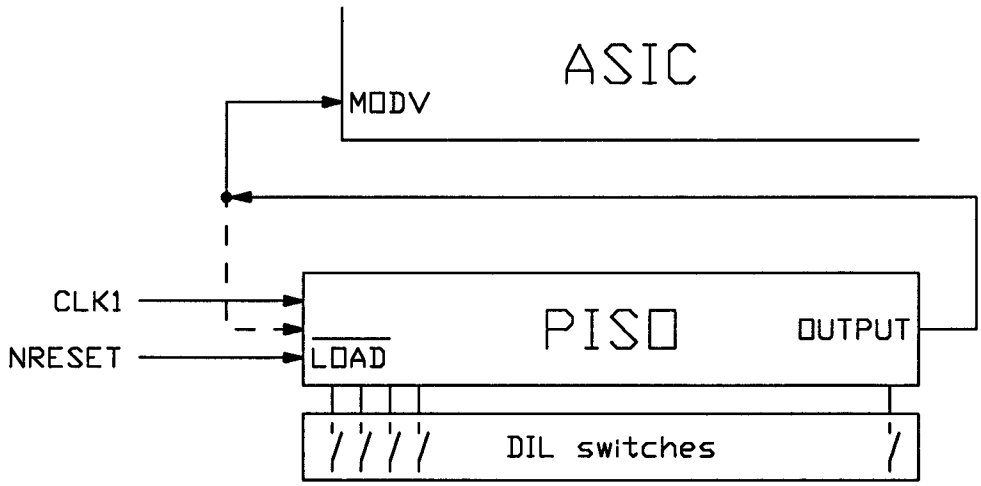


Figure 4.9: Supplying a modification vector

Adding a modification vector

Modification vectors are supplied in a serial manner through the MODV pin. A possible setup is shown in figure 4.9. The shift register is run by the input clock (CLK1) and the NRESET signal is used as load. If this signal consists of a single pulse then the shift register should be connected circularly. On the other hand, if a frame pulse is available the modification vector will be reloaded at the start of each code word and there will be no need for such connection.

4.2.5 Features left out of design

Although most of the initial requirements have been met, there are some features that would have been useful to include on the device. Shortage of space was the main reason why they were not.

The first, and most important one, would be another decision circuit to be optionally used instead of sub-modules WEIGHT and CMP4LE. This would make the decoder appropriate for decoding of burst-error-correcting codes as well.

The second would be a circuit to demultiplex the NERRDET output signal into two signals, ERRDET and ERCORR, for "ERRors DETected" and "ERrors CORRected". This would be better than sampling NERRDET using external logic.

Finally, one should note that the present circuit only implements a decoding scheme. It is assumed that a frame pulse and suitable clock signals can be made available, although it should be noted that use of runlength limited codes facilitates clock regeneration. Extra circuitry would therefore be needed to actually implement a receiver.

4.3 CAD tools

Together with suitable software, computers have been successfully utilised as design tools in many areas. Computer Aided Design packages offer the designer not only a means of entering the design onto the computer and obtaining hardcopies of it but also a whole set of editing facilities. The design may then be changed very easily, perhaps without even losing the previous version. Depending on the actual application area, other sorts of facilities such as simulation, auto-routing and compatibility with computer aided manufacture software may also be available.

This section is about a particular CAD software package for ASIC design, called BX. It is supplied by M.C.E. – Micro Circuit Engineering Ltd. and has been used to design, simulate and test the decoder presented in the last section. It implements a semi-custom design approach, where a circuit is built from standard library components (gates, flip-flops, etc.). The following subsections present the design procedure from the practical point of view of putting the design onto a computer and obtaining results from it. BX software runs on IBM compatible PCs.

4.3.1 Schematic capture

BX provides two ways of entering a design onto the computer:

Hardware description language The design is expressed in the form of descriptive text. It has the advantage of not requiring any graphics support but the design description may become very difficult to read, especially for someone who is not familiar with the language.

Schematic capture The design is drawn on the computer screen. Basic components are represented by graphic symbols and connections consist of lines going from one terminal to another. Schematic capture requires a graphics interface and a mouse to be attached to the computer but the final result is much easier to understand.

Schematic capture was chosen for this design. First, a small prototype for the error-trapping decoder module was developed, in order to test its functionality. Then, a full size module was designed. At this stage it became necessary to build registers and other large components and a hierarchical structure was constructed. At the lowest level standard library components can be found. One level upwards there are shift registers, comparators, multiplexers, counters and so on, each of which can be treated as a whole. These modules (together with some basic components) were then interconnected to obtain higher level modules of increasing complexity until the top level was reached. BX data files were named after PETLD for *Programmable Error-Trapping Line Decoder.*

The modular approach proved to be very useful as some of the larger components were already available from previous designs and were used with almost no alteration. Also, changes to a particular module are propagated to all copies of that module in the entire design.

Once the design has been fully entered it must go into a compilation process. The computer translates the graphic information into a more suitable format and ensures that several simple rules are respected, such as "all inputs must be connected to something" or "no outputs can be connected together".

PETLD design modules can be found in appendix A.

4.3.2 Design analysis and simulation

After successful compilation the design may be analysed in various ways. There are two major points of concern:

Fan-out Each gate is characterised by the number of inputs which its outputs are able to drive. Also, different inputs may present different loads to the output driving them. BX generates an extensive listing where a load factor is associated to each output in

the design. This load factor depends on the number and type of inputs connected to each output and should not exceed 10 for the majority of components provided. However, some inverting buffers may drive up to 16 inputs.

Fan-out check did reveal that some components in the design were overloaded, particularly those driving clock lines or controlling the first stages of large multiplexers. These problems were solved by inserting inverting buffers at strategic points, usually in pairs, in order to reduce the load each buffer had to drive.

Cell-usage As stated earlier the ASIC contains 1440 standard cells. However, MCE requests a maximum occupation of 80% (1152 cells) so that the design may be routed. The PETLD utilises 1245 cells (86.5%) which is over the limit. After considering the fact that it was not possible to decrease cell usage without losing functionality, it was decided to assume that the design *might* route. And it did...

Another important feature provided by BX software is simulation. BX libraries contain not only a logical description for each standard component (e.g., NAND, EXOR,...) but also a set of parameters associated to each of them. Parameters consist of typical time delays from the inputs to the outputs and eventually, as in the case of flip-flops, of typical set up and hold times. Thus, both the timing of the components and the logic function they implement are taken into account by the simulator.

Simulation requires the user to specify, by means of a text file (the test program), which signals should be considered as input and as output. External signals, which are those corresponding to the device pins, may be input, output or even bidirectional. Internal signals may be considered as outputs for simulation purposes but never as inputs. The PETLD has 35 input, only 2 output and no bidirectional signals. Next, the test program assigns to each input a specific waveform. Outputs to be monitored may also be assigned to waveforms, in which case comparison between the expected and the actual (simulated) values at each cycle is performed.

Simulation may be executed over typical or worse-case parameters. BX derives the latter from the former by a scaling operation. An important consideration lies behind this procedure, the fact that the fabrication process may present a wide range variation (from

30 to 170% of the typical values) but only over large silicon areas. For practical purposes it is reasonable to assume that all the cells in a particular chip have the same parameters. Temperature and supply voltage are other factors that influence speed of operation but, again, they are approximately the same for all the cells in a particular device.

Typical case simulation is usually run first, in order to check the functionality of the design. Only when it is working from the logical point of view are worst-case simulations performed, perhaps using another simulator feature, time checking. The next subsection goes through the test program that was written for the PETLD design and the one after that will be concerned with system characterisation, where the utility of the time checker becomes more evident.

4.3.3 Design testing

ASIC designs have to be tested according to two different points of view:

Functionality The goal is to make sure that the design produces the right results for a given problem. In the present case it means checking whether or not the circuit decodes given code words in the right way. It is assumed that eventual errors are due to the design itself, either because of errors in circuit entry or simply because the design is actually wrong.

Fault checking The assumption is now that the design is correct but one (or more) of its components may be faulty. This type of testing is oriented to the verification of manufactured chips and should ensure that *all* parts of the device are working properly.

From a certain degree of complexity on, it becomes impracticable to fully test a design. For combinational circuits the number of input vectors that would be necessary to cover all possibilities grows exponentially with the number of inputs, and the situation is a lot worse if circuits are sequential. For this reason, ASIC testing is a subject in itself and has been studied independently from the design. However, design-for-testing approaches where a circuit is designed not only to work but also to be easily tested seem to be the most reasonable.

The test program that was written for the PETLD is divided in 4 parts, as follows:

Hamming (7,4) This was the first part to be written. It presents to the design a series of Hamming(7,4) code words to be decoded (some of them with bits in error, or modified) and checks the results. The main purpose of this part was to test if the circuit was well conceived before expanding it.

BCH (31,16) Much like the previous case but for a code that uses the whole syndrome register (15 check bits) and with a greater error-correcting capability ($t = 3$).

Input and output register checking Written to check eventual faults in the input and output registers, this part disables the decoder module by setting the generator polynomial to all-zeros. The input and output registers are then fed with a sequence of alternate zeros and ones and its contents are checked using the `M` and `W` pins to propagate them to the output `DATAOUT`.

Syndrome register checking The condition "errors always trapped" is forced by setting $t = 15$. Then, the `J` pins are used together with the input signal to produce an "error pattern" that is added to the original code word and propagated to the output. This checks whether the syndrome register elements and respective feed-back connections are working.

All the input and output pins are exercised high and low, which should ensure that none of them will be stuck at one of the logic levels. This is a necessary condition to obtain a good fault coverage. Effort was made to vary circuit nodes as much as possible but since no systematic approach was used, it is difficult to express the achieved fault coverage in quantitative terms.

The actual test program can be found in appendix B. Its structure is presented in figure 4.10. The `*WAVE` statement gives the waveform a name and the `PERIOD` statement divides the time domain in slices of T nanoseconds. After that, up to 5 *timing generators* may be defined by means of `TG` statements. Timing generators will be associated to signals in order to tell the computer where inside each cycle transitions should occur (see figure 4.11a).

```
*WAVE design_name
 PERIOD T
 TGi t1 t2
*TYPES
 Signal_name I|O|C|N|etc.  TGj
*PATTERNS
/ !Group 1
 (Test patterns for group 1)
/ !Group 2
    ⋮
*END
```

Figure 4.10: Test program structure

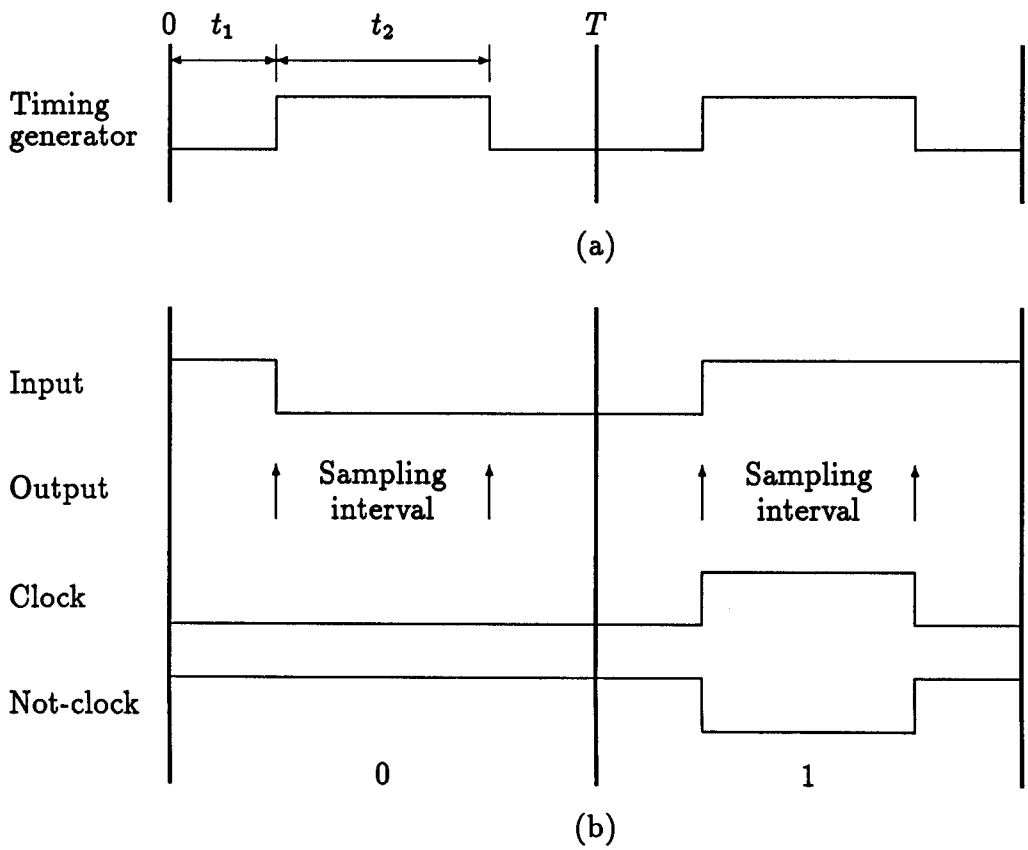


Figure 4.11: Timing generators and waveform types

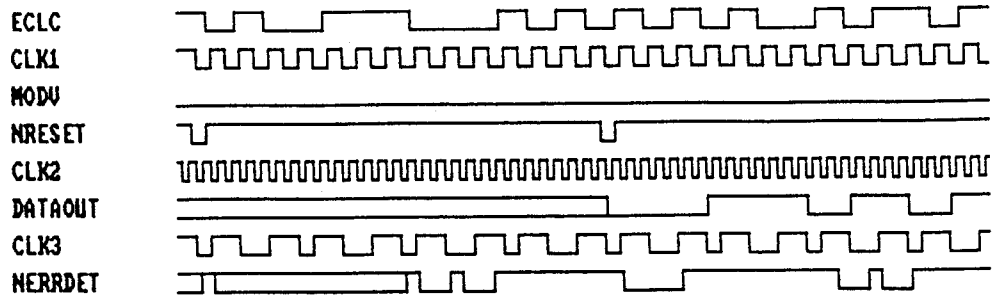


Figure 4.12: Simulated input and output waveforms

The *TYPES statement is followed by the definition of all waveforms to be generated. According to the type of waveform (Input, Output, Clock, Not-clock, etc.), t_1 and t_2 specify either where transitions occur or outputs are sampled (see figure 4.11b).

Finally, after the *PATTERNS statement each signal is described in terms of a sequence of zeros and ones (there is another symbol, X, that means "don't care" and may be used for output patterns only). BX provides several ways of defining patterns so that long sequences of the same symbol or periodic sequences may be defined in a compact way.

Because there is a limitation for the number of cycles the program is able to handle "at once", if one wants to use more than 256 cycles, waveforms have to be broken into groups. However, this is simply a programming constraint and the user is still able to deal with long patterns as if they were not broken at all.

Simulation results were visualised in graphic form by means of another BX facility called *display results graphically* (see figure 4.12).

4.3.4 System characterisation

After the design has been tested under the nominal conditions it is important to identify its characteristics, both static and dynamic. Static characteristics for the PETLD were obtained straight from the MCE documentation since they do not depend on the design itself. Dynamic characteristics, such as maximum clock speed, phase relationships between clocks, set up and hold times for the inputs and propagation delays for the outputs, had to be calculated. This was a very laborious job since BX only produces a delay listing, where propagation delays for each component are given for both positive and negative transitions.

It is up to the user to follow the circuit and sum all the delays up from one point to the other, in order to determine the appropriate parameters in each case.

First, upper bounds for the nominal parameters were determined. This was done considering that the propagation delay through a group of cascaded gates was the sum of the maximum delays in each gate. Yet, for simple cases account was taken of the signs of the transitions.

Then, worst-case parameters were derived from the nominal ones by scaling them by a factor of 1.7 (or 0.3, in the case of negative values). The complete data sheets for the PETLD device can be found in appendix C.

The simulator was finally utilised to check the applicability of such values. At this point the time checker was used to ensure that all the points in the circuit were stable at the end of each cycle and that timing constraints were respected for all of the circuit components.

4.3.5 Fabrication of device

Before sending the design to MCE a form had to be completed stating that some necessary design steps and precautions had been taken, e.g., the design does not include any pulse generators. All the applicable questions could be answered affirmatively.

A problem arose concerning pin employment. For economy reasons a 40-pin DIL package had been selected which should accommodate the PETLD 37 signal pins plus 2 supply pins. However, MCE usually uses 3 external pin pads to check silicon quality, thus reducing the number of pins available to the user. A by-pass of these tests was arranged with MCE and all 38 pins became available. The device pin out is presented in figure 4.13.

The design and test files were sent to MCE on a floppy disk along with the completed form and a listing of the desired pin out. There was still some concern about whether the design would route or not and also the understanding that by-passing some silicon tests would reduce device reliability.

Upon reception of a design, MCE tries routing it by means of an auto-router and simulates the routed version using the user's test program. Routing allows physical characteristics, such as track capacitance, to be taken into account and if this stage is passed

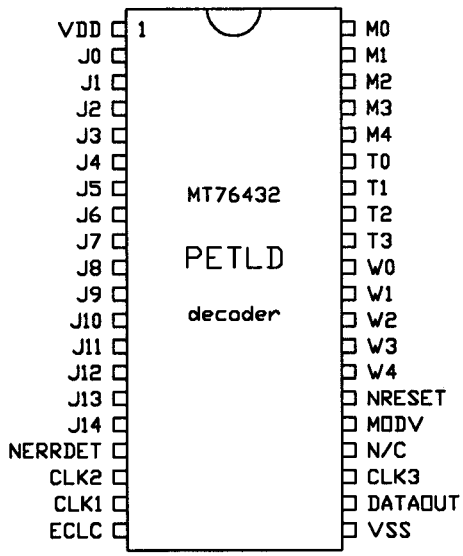


Figure 4.13: Device pin out

fabrication follows. MCE then tests the fabricated devices against the test program and dispatches them to the user.

Ten prototype devices were received from MCE under the chip reference MT76432 (figures 4.14 and 4.15). They were then verified as described in the next section.

4.4 ASIC verification

Verification of fabricated devices is a natural step to take before using them in some specific application. MCE by-passed some test procedures as discussed before and that meant devices could actually be faulty. Tests were carried out using a logic verifier, which provided a means of monitoring inputs and outputs while supplying appropriate test vectors to the device. No extra circuitry was needed.

4.4.1 The test equipment

The Tektronix LV500 logic verifier is composed of a terminal and a test head, both attached to a computer. The computer runs an application that enables the user to define test vectors and associate them with device pins via the terminal. It also applies those test vectors to the device under test (DUT) by means of the test head. There are several

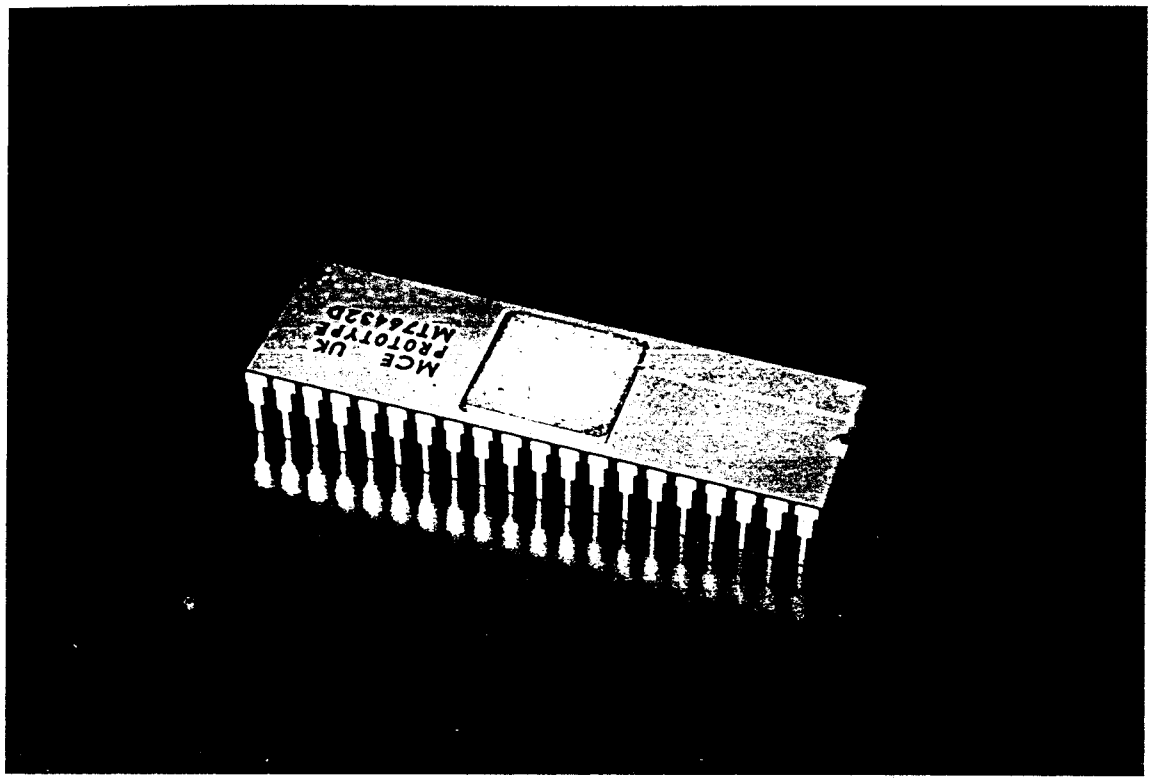


Figure 4.14: Fabricated device

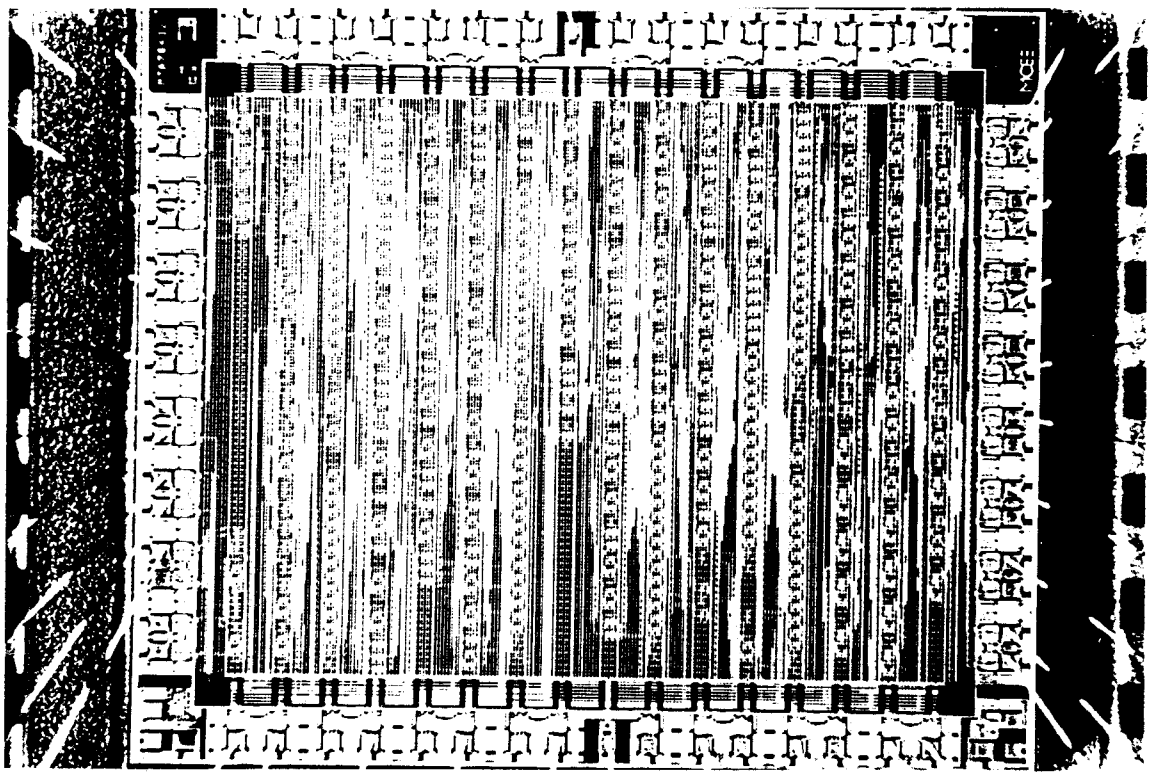


Figure 4.15: The MT76432 chip

facilities included such as editing, data format conversion, hardcopy generation and a few others.

BX simulation results were translated into a format that the LV500 could refer to by using home-developed software. The resulting waveform file was asynchronously sampled by the logic verifier to produce its own vector file. The LV500 was then configured with the device's pin names and the equivalent to BX's period and timing generators. A device was inserted in the test head and tests started.

4.4.2 Tests and results

The LV500 is capable of performing two types of test:

Pass/fail This is the first and simplest test to execute. The logic verifier applies the test vectors to the DUT and checks the results. This test was carried out over the ten prototypes and was passed by eight of them. The remaining two failed and were put aside.

Schmoo plots In these graphics, each axis represents a variable parameter of the device. For each pair of values a pass/fail test is run and the result is plotted accordingly. These plots appear divided in two regions, pass and fail, and give a good image of how the device works under a wide range of conditions. However, they take considerable time to produce.

Schmoo plots were constructed for one of the eight prototypes that passed the previous stage and are presented next.

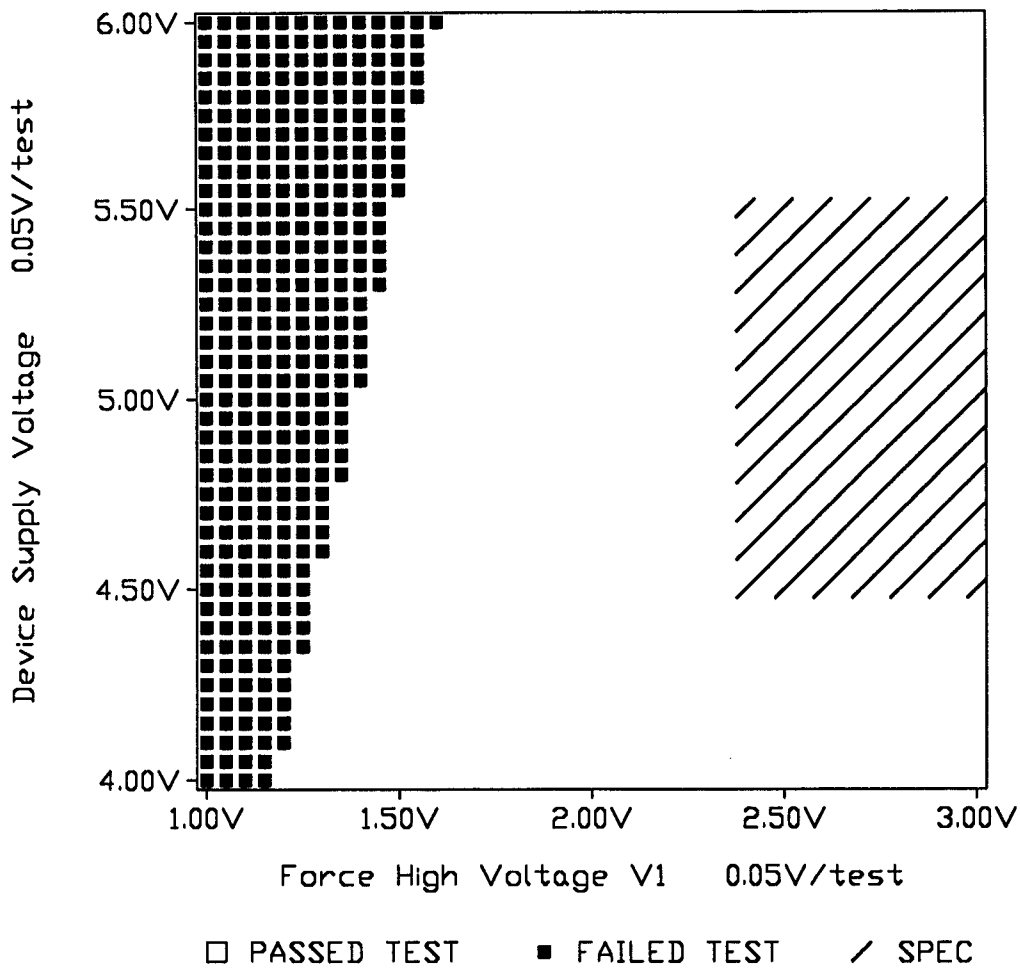


Figure 4.16: Device supply voltage vs input high threshold

Schmoo plot 1: Device supply voltage vs input high threshold

This plot shows the minimum input voltage that constitutes a logic "1" against the device supply voltage. When the input high voltage falls below the threshold the device fails. Comfortable performance over device specifications (appendix C) can be observed (shaded zone).

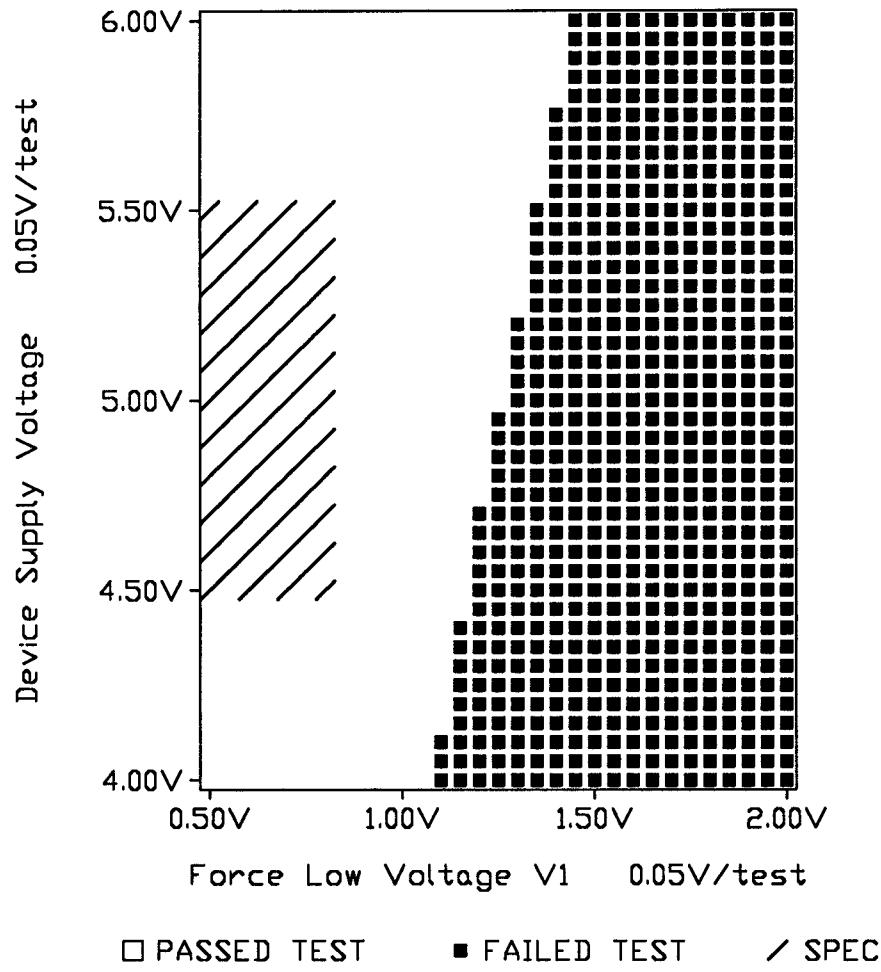


Figure 4.17: Device supply voltage vs input low threshold

Schmoo plot 2: Device supply voltage vs input low threshold

Like the previous plot, but for the maximum input voltage that constitutes a logic "0". Again, the device outperforms its specifications.

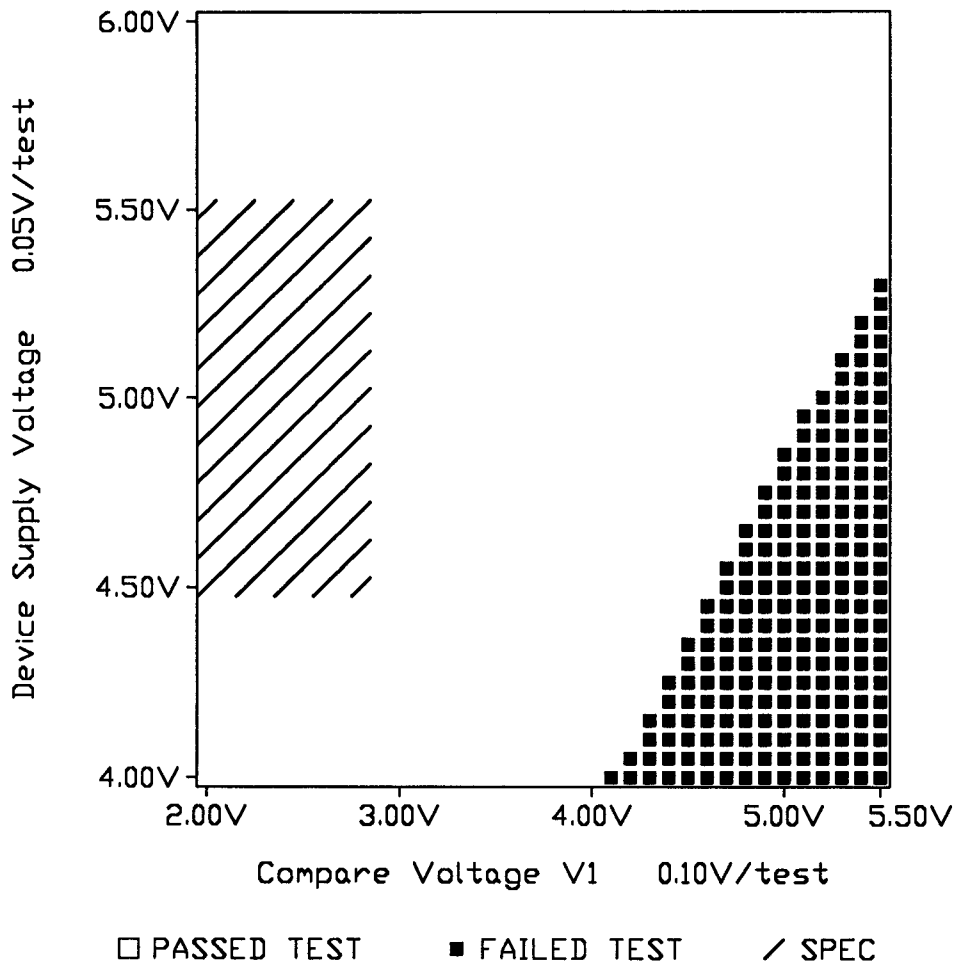


Figure 4.18: Device supply voltage vs output high threshold

Schmoo plot 3: Device supply voltage vs output high threshold

This plot shows output high voltage against supply voltage. The device fails when the verifier threshold goes beyond the device output level. A direct relationship between the two variables is observed, $V_{OH} \simeq V_{DD}$. This relation suggests that the LV500 does not impose heavy loads to DUT outputs. Since the current actually debited by the device outputs could not be determined, care must be taken when comparing measured performance and device specifications. In fact, these are defined for an output current I_{OH} of 1mA.

No plot was constructed for the output low threshold because the minimum compare voltage provided by the LV500 is 0.5V, too high when compared to the TTL low level.

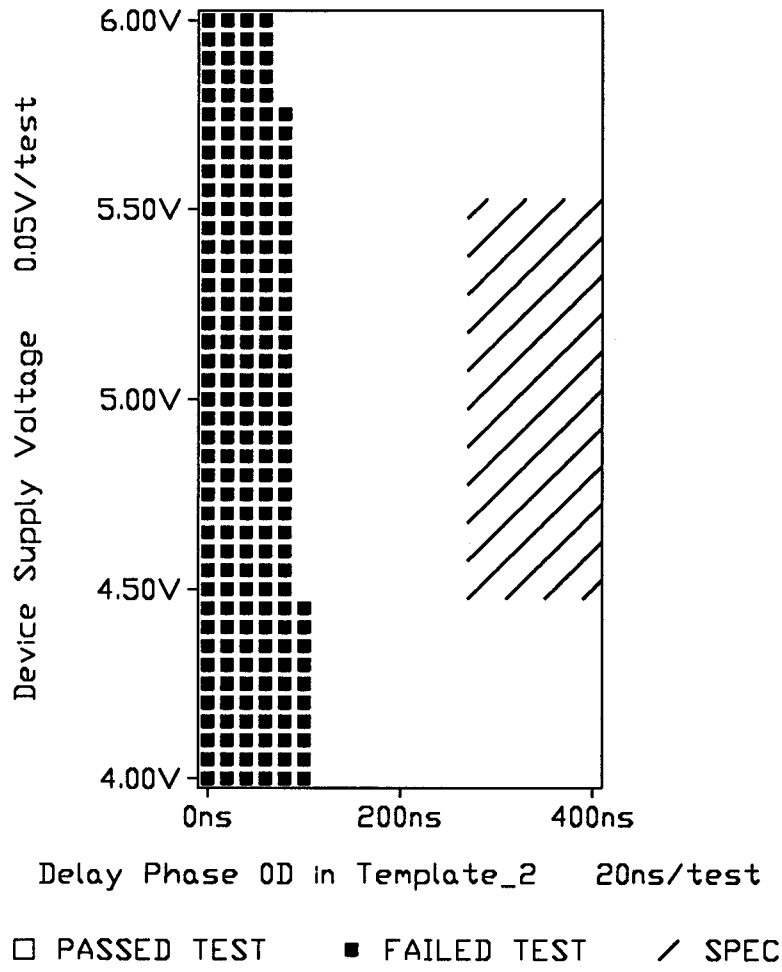


Figure 4.19: Device supply voltage vs output delay (DATAOUT)

Schmoo plot 4: Device supply voltage vs output delay (DATAOUT)

This plot shows how supply voltage affects device speed. The propagation delay is measured from a falling edge of CLK3 to the moment data becomes valid on DATAOUT. As expected, the device is faster for higher supply voltages. The measured delays are clearly less than the worst case value, which can be found in the data sheets. Although the LV500 presents DUT outputs with less than 35pF load (according to the manuals) and specifications assume 100pF, one should note that this affects the output buffers only. The delay in the output buffer is just a fraction ($\sim$ 20%) of the overall delay, therefore comparison between nominal and measured performance is still reasonable.

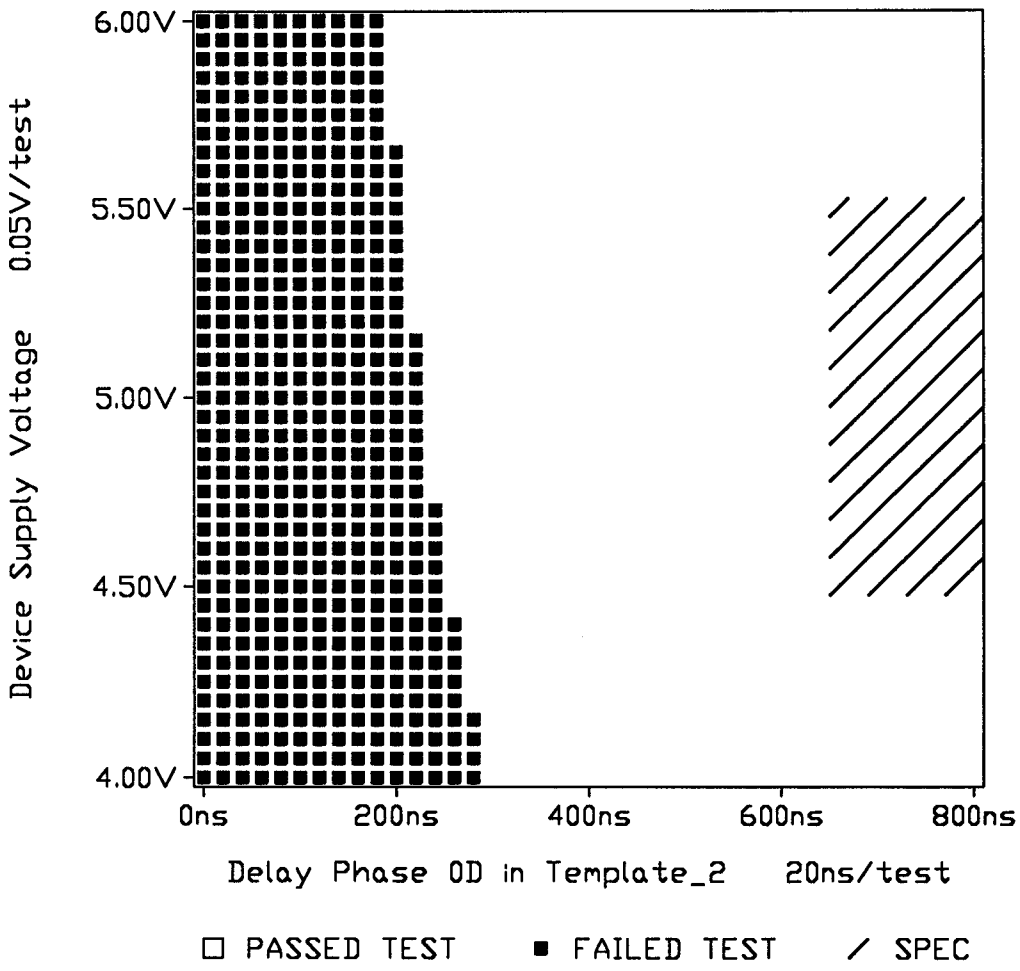


Figure 4.20: Device supply voltage vs output delay (NERRDET)

Schmoo plot 5: Device supply voltage vs output delay (NERRDET)

Like the previous plot but for the NERRDET output. In this case the delay is measured from the falling edges of CLK2. Considerations made in the last subsubsection about output loads and propagation delays apply to an even greater degree here, as the overall delay is longer.

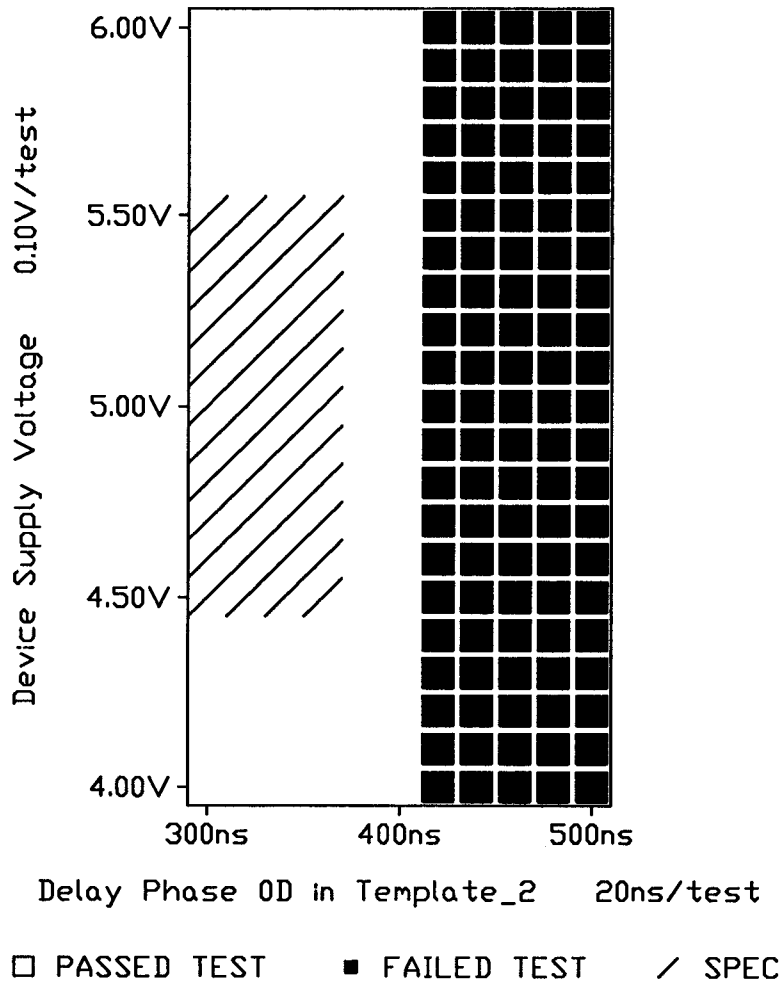


Figure 4.21: Device supply voltage vs (400ns $- t_{\text{setup}}$) (ECLC)

Schmoo plot 6: Device supply voltage vs set up time

This plot shows the minimum amount of time required for data to be present at the ECLC input before a negative transition occurs on CLK1. The reason why the graph is inverted and offset has to do with the LV500 timing generators. Set up time does not seem to vary with supply voltage but any variation less than 20ns would have been missed at this scale (maximum resolution available).

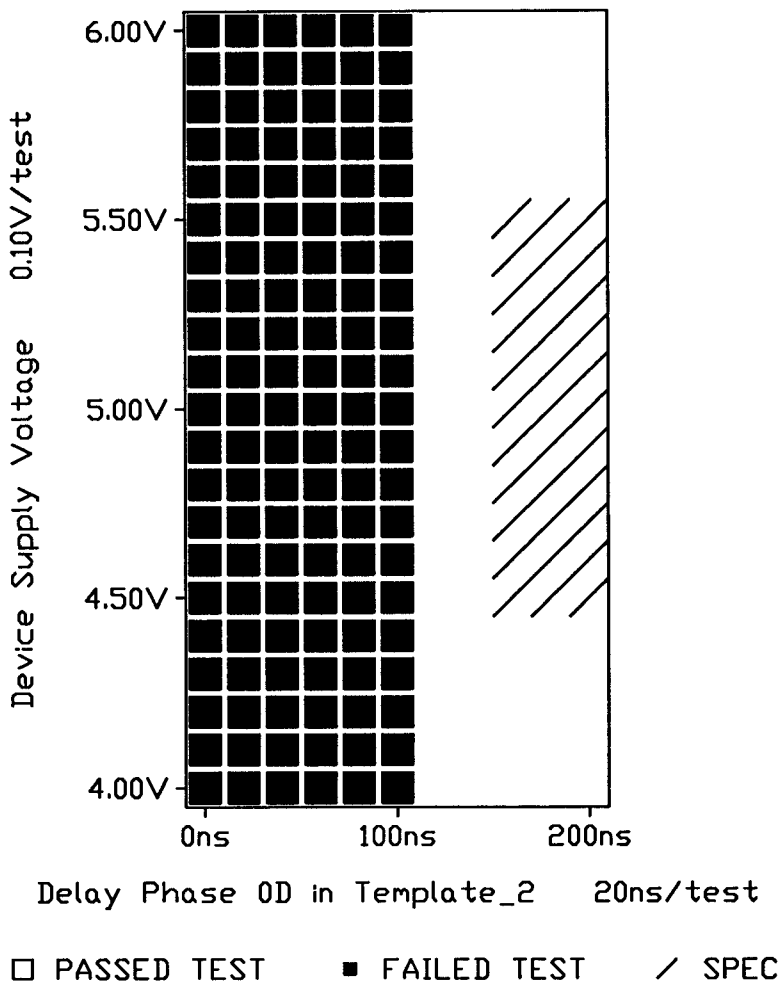


Figure 4.22: Device supply voltage vs (100ns + t_{hold}) (ECLC)

Schmoo plot 7: Device supply voltage vs hold time

Like the previous plot but for the minimum time data is required to remain valid on the ECLC input after a negative transition has occurred on CLK1. This graph is not inverted but is still offset.

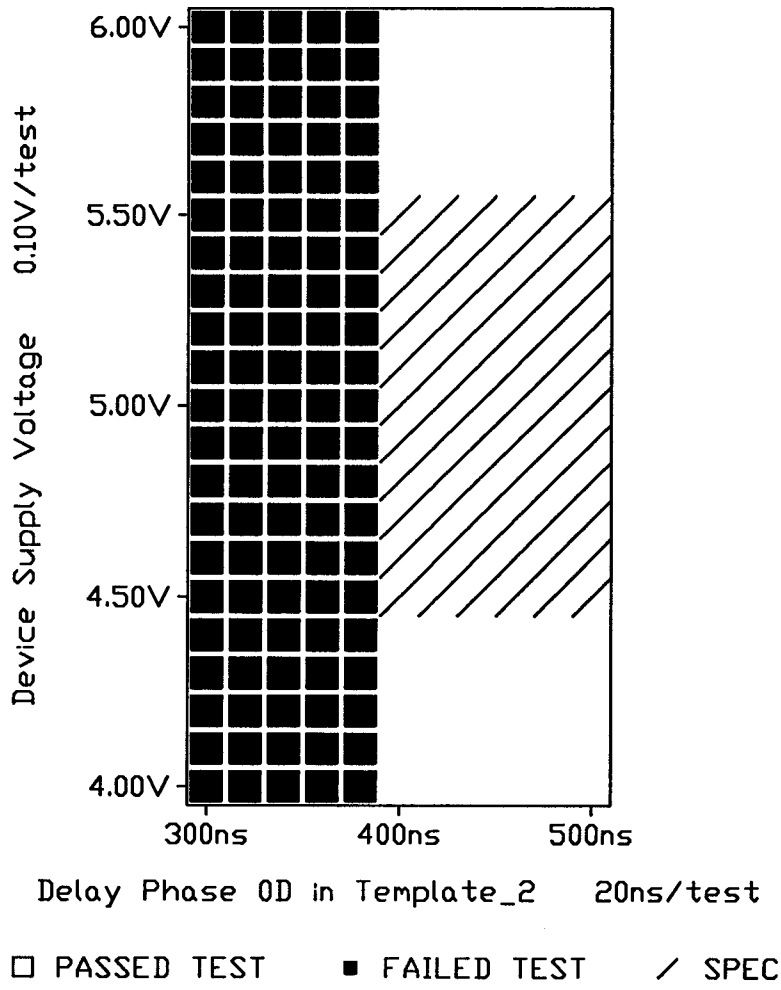


Figure 4.23: Device supply voltage vs (400ns + t_1)

Schmoo plot 8: Device supply voltage vs clock phase (t_1)

This plot shows the minimum amount of time required from the negative transition on CLK2 to which the other signals relate (see figure 4.7) to the corresponding negative transition on CLK1. Again, a better resolution would be convenient. Device specifications are still met.

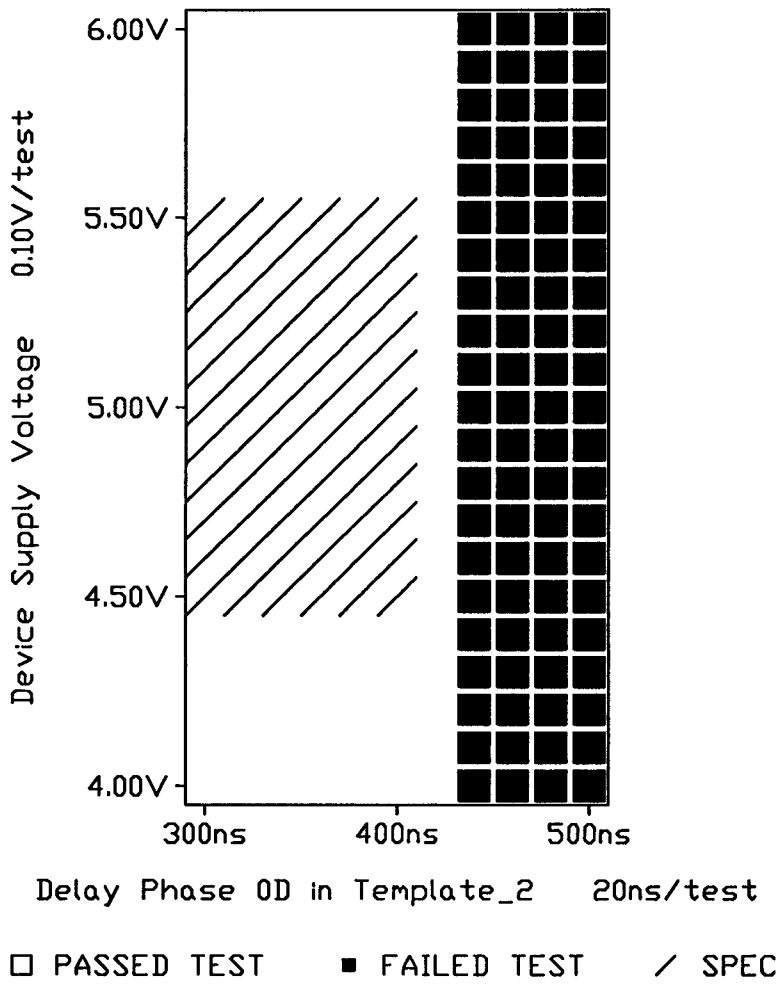


Figure 4.24: Device supply voltage vs (400ns $- t_3$)

Schmoo plot 9: Device supply voltage vs clock phase (t_3)

Like the previous plot but for the time from the negative transition on CLK3 to the reference transition on CLK2. The graph is inverted again.

4.5 Summary

This chapter presented the PETLD decoder and described its implementation in ASIC technology. Device specifications were drawn up and a modular description was given for the design. Then, hardware and software tools that had been used during device design and verification were introduced.

Eight out of ten fabricated devices passed the tests stage. One of these was investigated more deeply in order to get some idea of how the PETLD behaves under extreme (non-nominal) conditions.

A single device was later used in conjunction with Hutchinson's [5] PECLC encoder to implement a runlength limited coding scheme. This will be the subject of the next chapter.

Chapter 5

System Proving and Results

5.1 Introduction

The PETLD chip was used to experimentally prove that runlength limited codes have the same error-control power as the standard codes they are derived from. The experiment is described in the next two sections. Section 5.4 presents the final results.

5.2 Experiment description

In order to measure the improvement in error-rate performance obtained by using a RLECC coding scheme, a simple communications system was simulated (see figure 5.1).

Data from a PRBS source was input into the communications channel, either directly to calibrate the system or through the encoder chip, according to the switch position. Channel simulation consisted of adding Gaussian band limited white noise to the flowing data. The transmitted signal was then latched, decoded (if appropriate) and monitored by an error-detector. Different channel bit-error-rates were obtained by varying the noise level.

For the purpose of this experiment, a BCH(15,5) based RL code was selected. Although being a triple-random-error correcting code, the BCH(15,5) can be decoded using error-trapping at a small penalty in error-performance when compared to MLD. In fact, all single and double errors and most combinations of triple errors (450 out of 455) can be corrected.

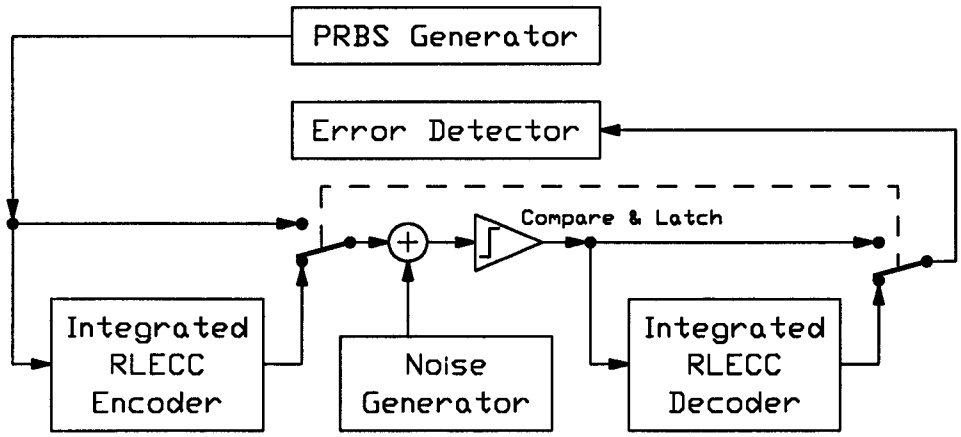


Figure 5.1: Test set up

The code runlength was limited to 10 by choosing an appropriate modification vector. Popplewell [1] states a minimum runlength of 6 for BCH(15,5) based codes but that bound can only be reached if non-cyclic versions of the parent code are considered, which is not the case here.

5.3 Design description

Despite the simplicity of the block diagram in figure 5.1 and although encoding and decoding could be performed by single integrated circuits, it was still necessary to provide the external logic and generate the control signals those circuits require. The design was split into four modules which can be understood separately. The following subsections describe each of them in some detail. Schematics and other specific information about the design are given in appendix D.

5.3.1 Encoder

The PECLC is a flexible runlength limited error-control encoder. It was designed and tested by A. Hutchinson at the University of Wales, Bangor, and fabricated by M.C.E. under the reference MT76348. It may be configured to generate any cyclic code within the same bounds as the PETLD and performs runlength limitation also by addition of externally supplied modification vectors. Figure 5.2 presents its pin out.

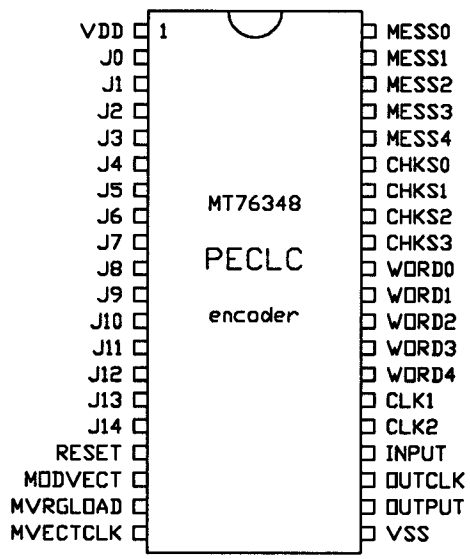
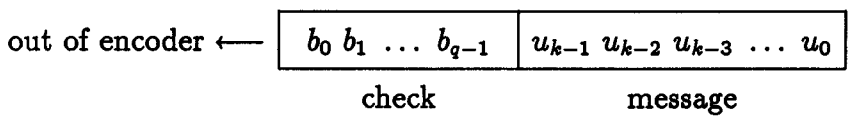


Figure 5.2: PECLC encoder pin out

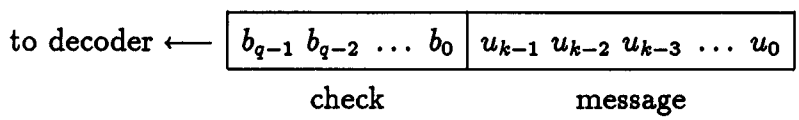
Although the PECLC chip was designed to generate cyclic codes, it does not actually produce code words in cyclic form. In fact, output code words present an inversion in the order of the check bits, as shown in figure 5.3. This difference in format may be irrelevant in terms of encoding (or not even that, as extra logic was needed to maintain flexibility) but raises serious difficulties for the decoding process. Check-bit order inversion was regarded as a 'bug' when designing the PETLD chip, which does assume that original (non-modified) code words are in cyclic form.

The encoder module (figure 5.4) includes not only the PECLC chip and the additional logic it normally requires but also an extra circuit to restore the cyclic code word format. The chip was configured to generate the BCH(15,5) by hard-wiring its J, MESS, CHECK and WORD pins.

Modification vectors were supplied to the encoder by means of a PISO register. Different vectors can be obtained by setting the parallel inputs of the register to the appropriate states, high and low. A set of 15 DIL-switches was used for this purpose. Switches were wired such that the referred difference in code word format was transparent to the user.



(a) Actual encoder output format



(b) Decoder input format

Figure 5.3: Code word formats

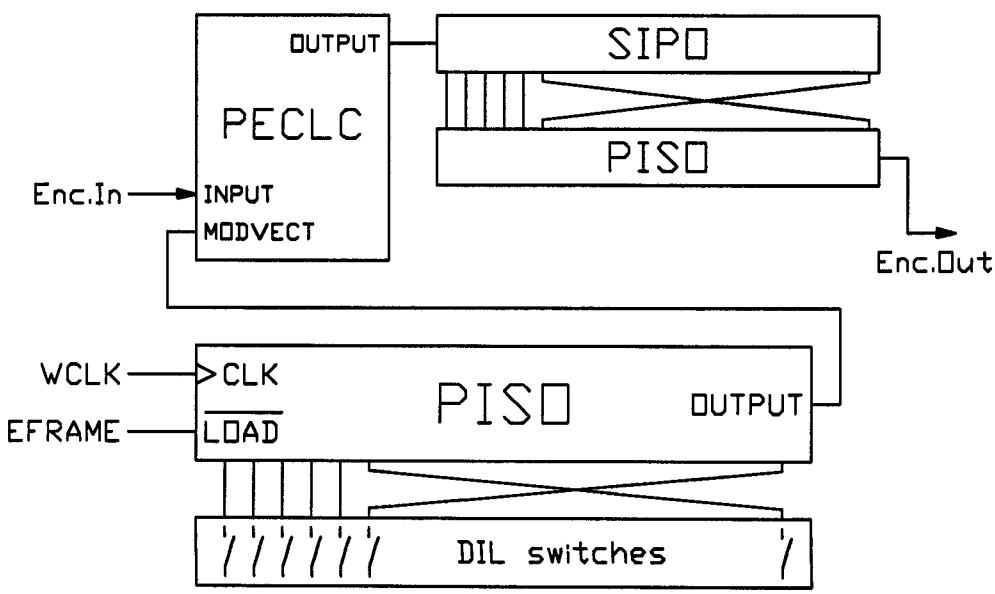


Figure 5.4: The encoder module

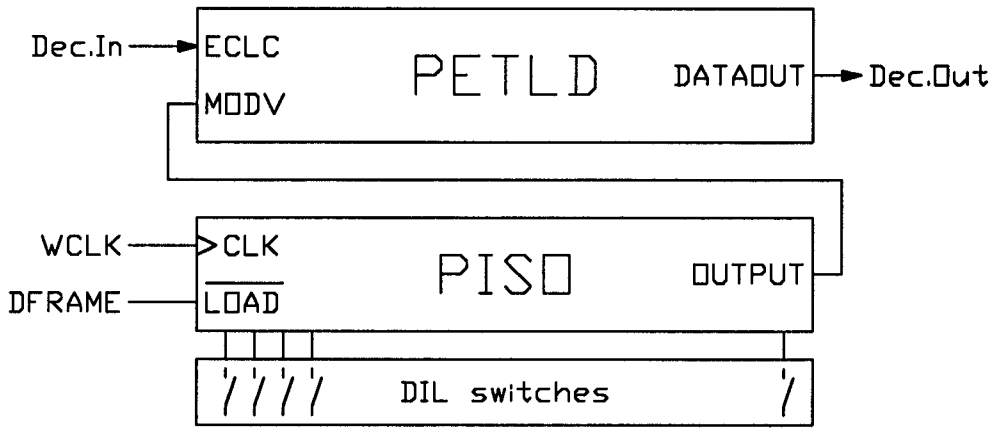


Figure 5.5: The decoder module

5.3.2 Decoder

The decoder module (figure 5.5) consists essentially of a PETLD device, configured in the same way as the encoder. Parameters for the BCH(15,5) and the corresponding configuration settings for the PETLD can be found earlier in subsection 4.2.4.

The NERRDET pin was not used, which means that no information about error-detection was taken into account. This procedure conforms with the channel being unidirectional.

5.3.3 Control signals generation

Although the encoding and decoding processes are concentrated on the PECLC and the PETLD, both devices require a set of control signals to be supplied externally, in particular clock and initialisation signals. These are in general code dependent.

Three clock signals were required:

MCLK The message clock. This signal defines the message bit-rate and corresponds to CLK1 on the PECLC and to CLK3 on the PETLD.

WCLK The code word clock. It defines the code word bit-rate and corresponds to PECLC's CLK2 and to PETLD's CLK1.

DCLK This signal corresponds to the decoder CLK2. It is not used by the encoder.

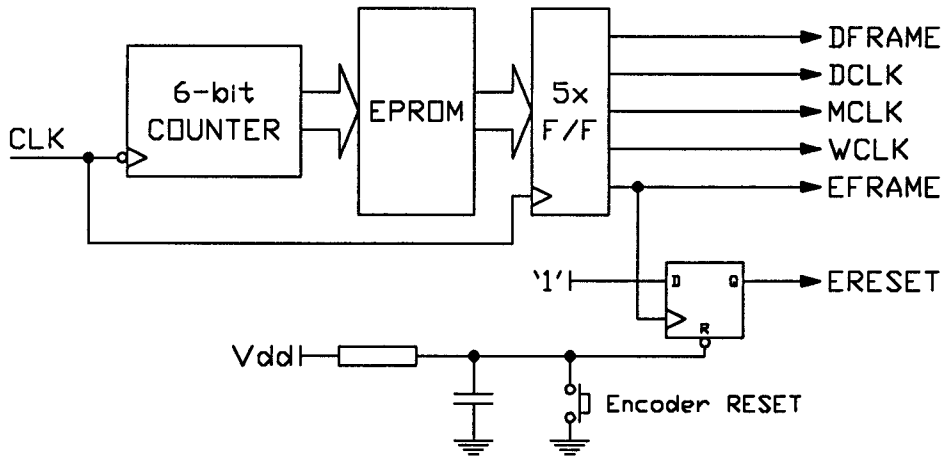


Figure 5.6: The control signals generator

For the BCH(15,5) case, clock signals should be related in frequency as follows:

$$f_{\mathrm{DCLK}} = 2 f_{\mathrm{WCLK}}$$
$$f_{\mathrm{MCLK}} = \frac{5}{15} f_{\mathrm{WCLK}} = \frac{1}{3} f_{\mathrm{WCLK}}$$

Instead of using frequency dividers to obtain MCLK and WCLK from DCLK, patterns for each signal were stored in an EPROM (figure 5.6). This procedure made it very easy to ensure that transitions occurred at the right time. Scanning the EPROM contents with a simple counter and latching its outputs would reproduce the stored patterns in the time domain.

Using an EPROM also facilitated the generation of initialisation signals. The PECLC requires a *single* negative pulse to be applied to its RESET pin, until after the 'first' positive transition in MCLK. On the other hand, the decoder requires a negative pulse (which can be periodic) to be applied to its NRESET pin until after the negative edge of DCLK to which the other signals refer. Further, both devices needed to be initialised in a synchronised form.

The patterns that were used as control signals can be found in figure 5.7, which also presents the timing for the rest of the circuit. On the encoder side, message and modification bits become valid at the positive edges of MCLK and WCLK, respectively, and are clocked in the PECLC on the corresponding negative edges. Code word bits are output on the negative edges of WCLK and input to the pair SIPO/PISO in charge of altering the code word format.

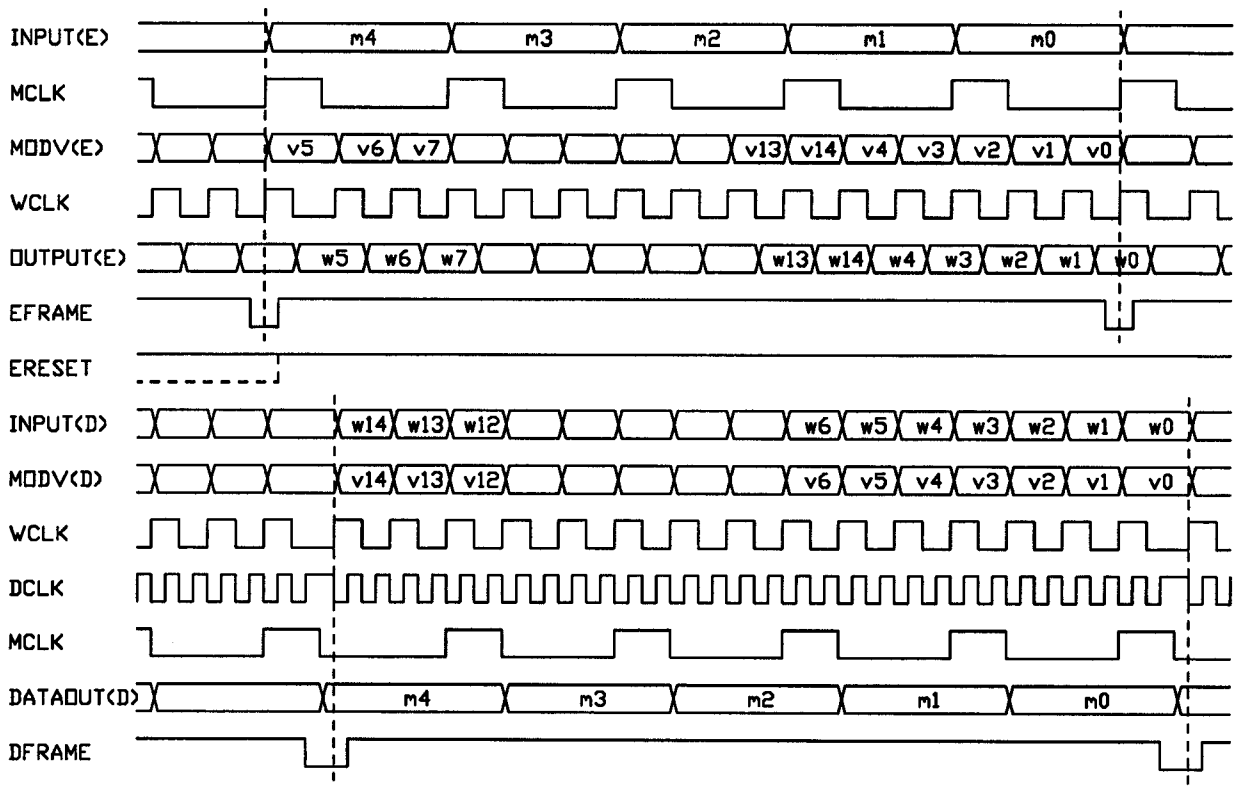


Figure 5.7: Circuit timing

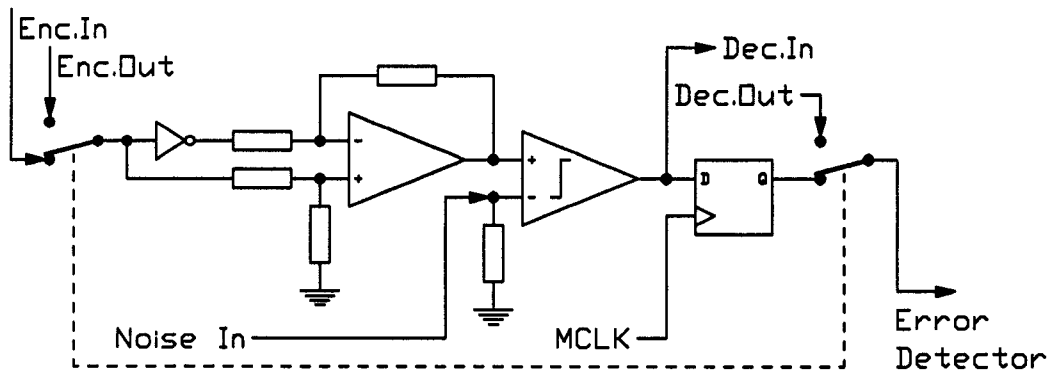


Figure 5.8: The channel error simulation circuit

The EFRAME signal is used both to generate the encoder reset signal ERESET and to load the modification vector into the PISO register. The PECLC generates its own signals to drive the modification vector register, but they assume asynchronous loading and were not used for that reason.

On the decoder side, code word and modification vector bits are clocked in on the negative edges of WCLK. Another frame signal, DFRAME, is used to initialise the device and, in conjunction with WCLK, to drive the modification vector register. Decoded message bits are clocked out on the negative edges of MCLK.

Note the phase relationships between the three clock signals, which satisfy both the encoder and the decoder requirements.

5.3.4 Simulated channel

This module (figure 5.8) simulates the behaviour of a memoryless symmetric communications channel. Symmetry is achieved by converting the TTL levels into a bipolar format. This task is performed by an OpAmp wired as a differential amplifier.

A voltage comparator was used to restore the TTL levels and, simultaneously, to add Gaussian bandlimited white noise to the transmitted signal. In fact, comparing the original signal to the instantaneous noise level has exactly the same effect as adding noise to the signal and comparing the result with zero. Varying the noise level will affect the channel error-rate.

At this stage the signal is latched, either by the decoder or by a flip-flop, according to

the switch position. The switch defines two modes of operation:

Calibration Encoder and decoder are by-passed. The bit-error-rate at the output corresponds to the channel error-rate.

Measurement Encoder and decoder are inserted in the data path. The bit-error-rate at the output is the residual error-rate and corresponds to those errors the decoder could not correct.

Since no reactive components were used, it is reasonable to assume that the simulated channel has no memory. The whole circuit was run at about 20Kbps (WCLK) in order to minimise the effect of rise and fall times at the amplifier and comparator outputs, and to ensure that little correlation existed between consecutive decisions (the noise bandwidth was only 50KHz).

5.4 Results

The test circuit was mounted on a printed circuit board and wire-wrapped. This was a laborious task but had the advantage of making the circuit easy to debug and even change, if necessary. A logic analyser was used to ensure that all modules were working properly.

After debugging, the circuit was connected to a transmission analyser and a noise generator. The analyser was configured to generate (and monitor) a pseudo-random binary sequence ($2^{31} - 1$) while the noise level was varied to check its influence on the channel error-rate. The chosen modification vector was then programmed by means of the DIL switches.

Measurements were performed in two steps, the first of which consisted of calibrating the simulated channel for a particular error-rate, by varying the noise level. After obtaining readings for that error-rate, the circuit was switched to 'measurement' and the value for the corresponding residual error-rate was taken.

The plot in figure 5.9 shows the relationship between channel error-rate and the achieved residual error-rate, when compared to theoretical results. The solid line represents the error-rate performance of the standard BCH(15,5) code when decoded by minimum distance (or maximum likelihood, MLD) [6]. If error-trapping is used instead, a slightly worse

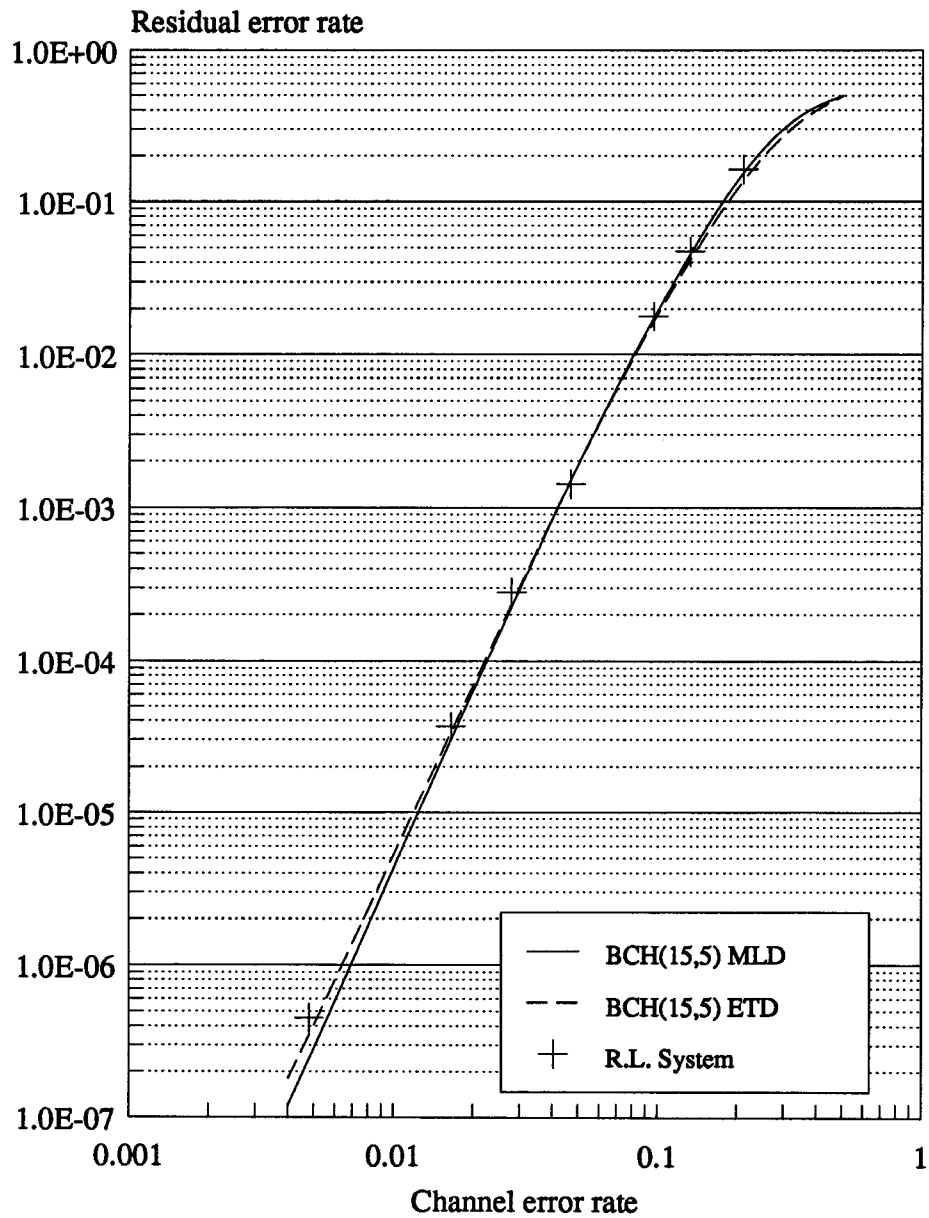


Figure 5.9: Bit-error-rate performance of runlength limited BCH(15,5)

characteristic is obtained (dashed line), indicating that this sub-optimal decoding method incurs only a marginal penalty in error performance. Experimental data ('+' signs) follow the theoretical results very closely, thus proving that runlength limited BCH(15,5) has indeed the same error-control power as its standard version.

5.5 Summary

In this chapter, the implementation of a runlength limited error-control coding scheme working over a simulated communications channel was described. The system was built in order to measure the error performance of runlength limited BCH(15,5).

Experimental results were obtained from the implemented system, which prove that RL codes maintain the same error control as their parent codes. Further, since the experimental data conforms to what was expected from the theory, functionality of the PETLD chip was also proved.

Chapter 6

Conclusion

This dissertation has been concerned with the implementation of a decoding device for a class of runlength limited error control codes which could be used in conjunction with an encoder designed previously (the PECLC).

After presenting the necessary background, the ASIC implementation of such a decoder was considered. The design was described in terms of functional modules and the software tools used were presented. Fabricated devices were verified and tested against the design specifications.

Finally, a runlength limited error control coding scheme working over a simulated communications channel was implemented, making use of the encoder and decoder prototype devices available. A BCH(15,5) based RLECC was used to experimentally prove both the error rate performance of RLECCs and the functionality of the device designed.

The results show that the fabricated PETLD does indeed operate according to the specifications, being easily configurable and providing error performance as predicted by the theory. The device can be used for the decoding of both runlength limited and standard cyclic error control codes and would interface directly with the PECLC if non-modified code words generated by the encoder were cyclic.

Fabricated devices performed comfortably beyond specifications, both static, which were supplied by MCE, and dynamic, which were calculated theoretically. Two out of ten prototypes were found to be faulty and were put aside. Although faulty devices have not been investigated yet, it was noticeable that they failed in different regions of the test

program.

The PECLC and the PETLD chips constitute a pair of hardware tools which can be used in the future to investigate RLECCs more deeply. However, correcting the code word output format of the PECLC as well as giving the PETLD the ability to decode burst-error-control codes would be particularly appropriate, especially if improved technology versions of either chip were to be designed.

The PECLC/PETLD pair could be used to compare the error performance of RLECCs to that of standard ECCs in systems where timing is recovered from the data stream, at the receiver end of the channel. The two chips could also be used as building blocks for combined dc-free error control code encoders and decoders, as described in [2]. Although binary dc-free codes have implicit limited runlength, the runlength bound can be tightened by using RLECCs instead of standard ECCs to generate them. Codes resulting from combining the two line coding approaches would realise further improvements in low frequency attenuation.

In summary, a flexible ASIC decoder for a new class of runlength limited error control codes was designed, fabricated and tested satisfactorily, and used to prove experimentally the error control performance of such codes.

References

[1] Popplewell, A. and O'Reilly, J.J., "Runlength Limited Binary Error Control Codes" (under review).

[2] Popplewell, A. and O'Reilly, J.J., "Class of Disparity Reducing Transmission Codes with Embedded Error Protection", *IEE Proc.*, Vol. 137, Pt. I, No. 2, April 1990.

[3] Lin, S. and Costello, D.J., Jr., *Error Control Coding: Fundamentals and Applications*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1983.

[4] Fonseca, C.M., Williams, S. and O'Reilly, J.J., "VLSI Implementation of a Runlength Limited Error Control Coding Scheme", Third Bangor Symposium on Communications, Bangor, U.K., May 1991.

[5] Hutchinson, A.K., *A Flexible Runlength Limited Error Control Encoder ASIC for Data Storage and Transmission*, B.Sc. Honours Dissertation, University College of North Wales, Bangor, 1990.

[6] Olaniyan, O.O. and Turner, L.F., "On the error-correcting capability of optimum linear block codes", *IEE Proc.*, Vol. 123, No. 1, January 1976.

[7] BX Software Handbooks.

[8] Tektronix LV500 Manuals.

Appendix A

PETLD Design Modules

In this appendix the complete schematics for the PETLD (MT76432) chip are presented.

A.1 Design hierarchy

As stated in section 4.3, the PETLD design was entered into the computer by means of the schematic capture facility of the BX software package. In order to make it easier to understand and debug the design was modularised. The modular approach also proved to be useful since modules which were available from previous designs could be re-used with almost no alteration. This is the case for some large registers and multiplexers.

The circuit is composed of a total of twenty modules which are organised hierarchically as shown in figure A.1. The root module is called SHEET1 and looks much like a block diagram. Input and output pads correspond to the device pins.

One level downwards are the four modules described in subsection 4.2.3. These are represented by blocks in the root sheet and detailed in separate sheets, being composed of lower level modules which perform simpler operations. Inputs and outputs of non-root modules correspond to internal connections on the chip.

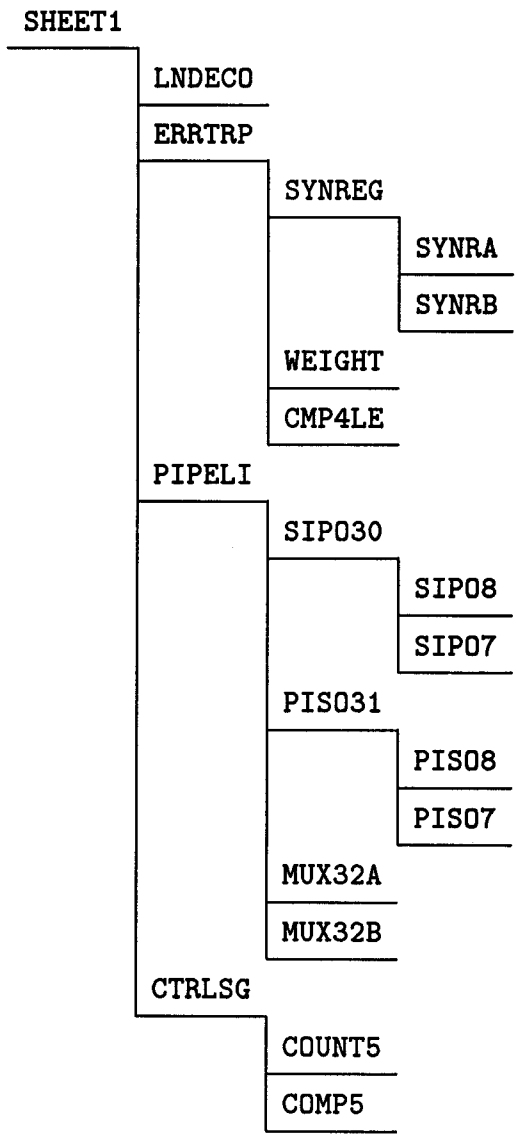
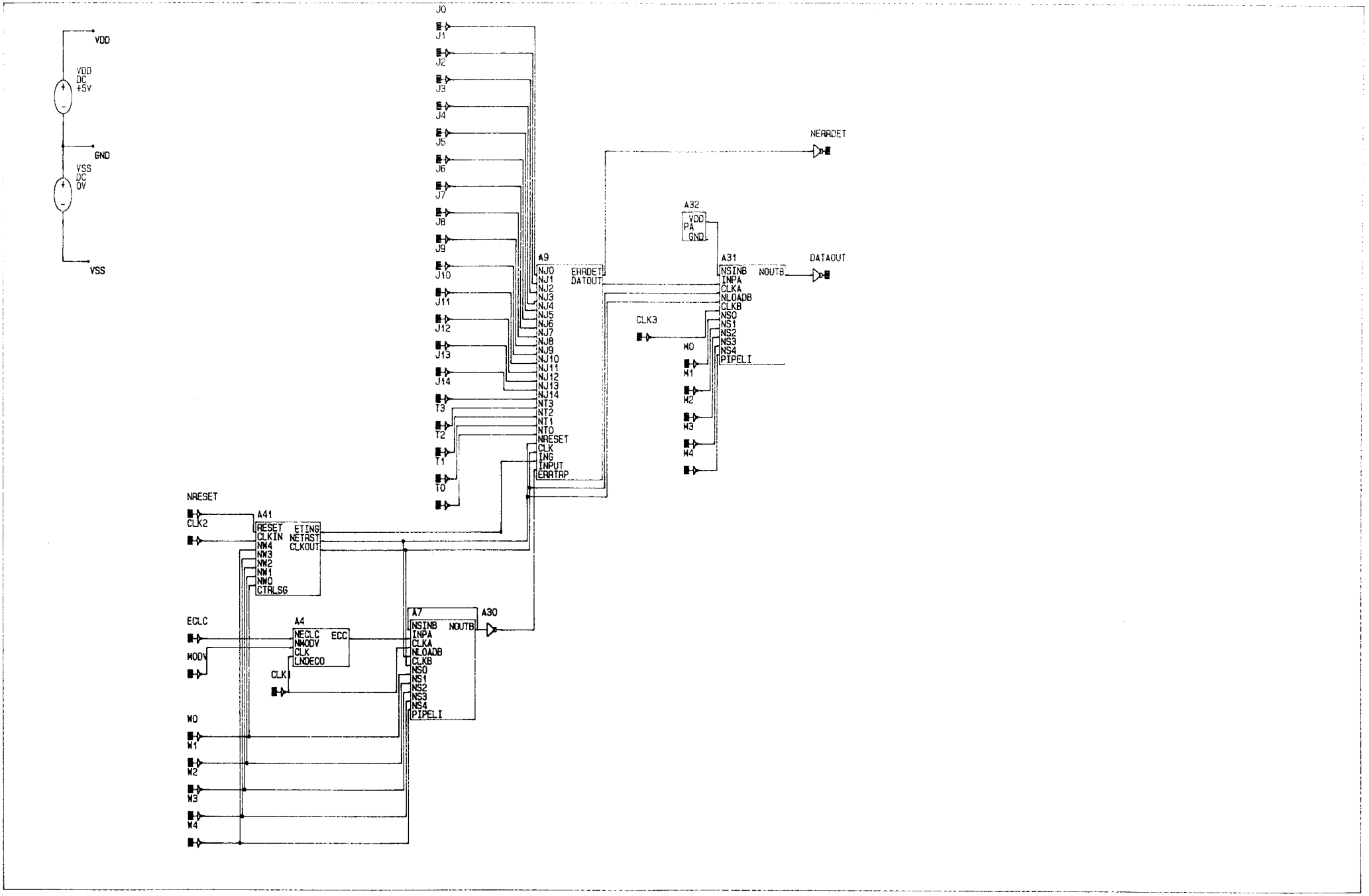
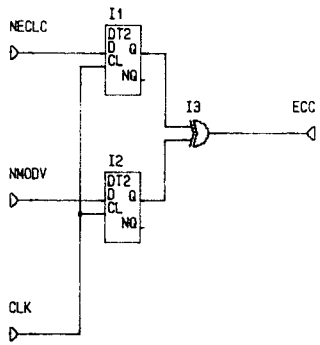


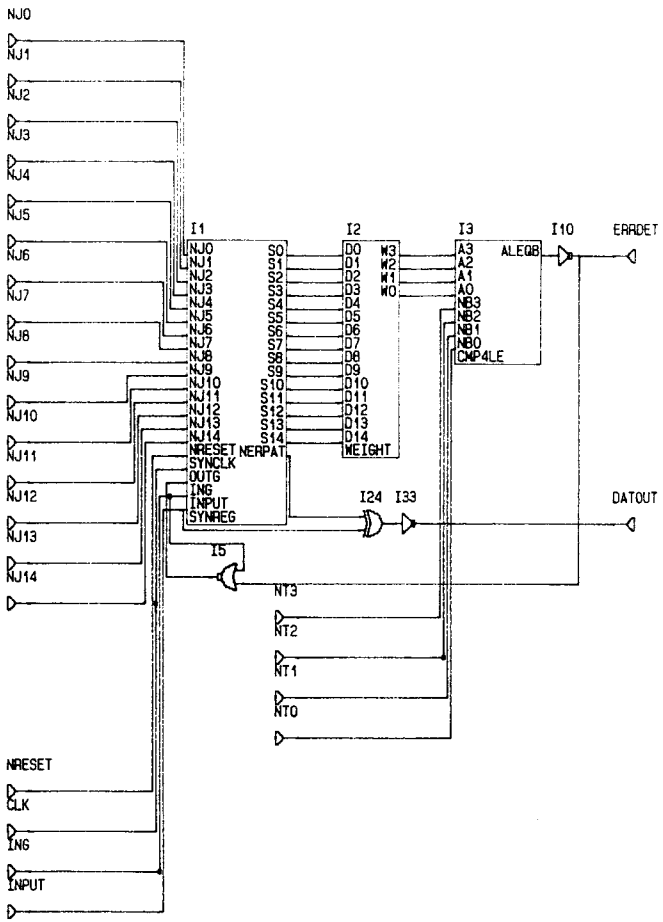
Figure A.1: PETLD modular hierarchy

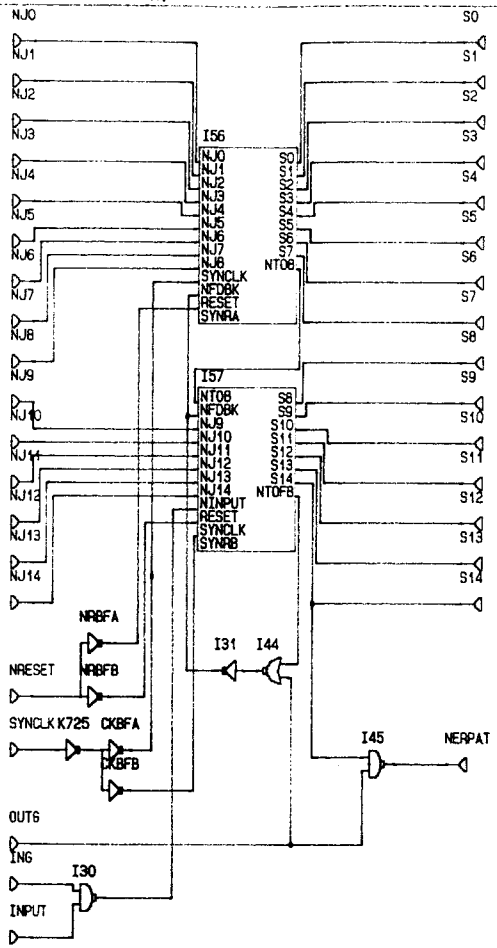
A.2 Schematics

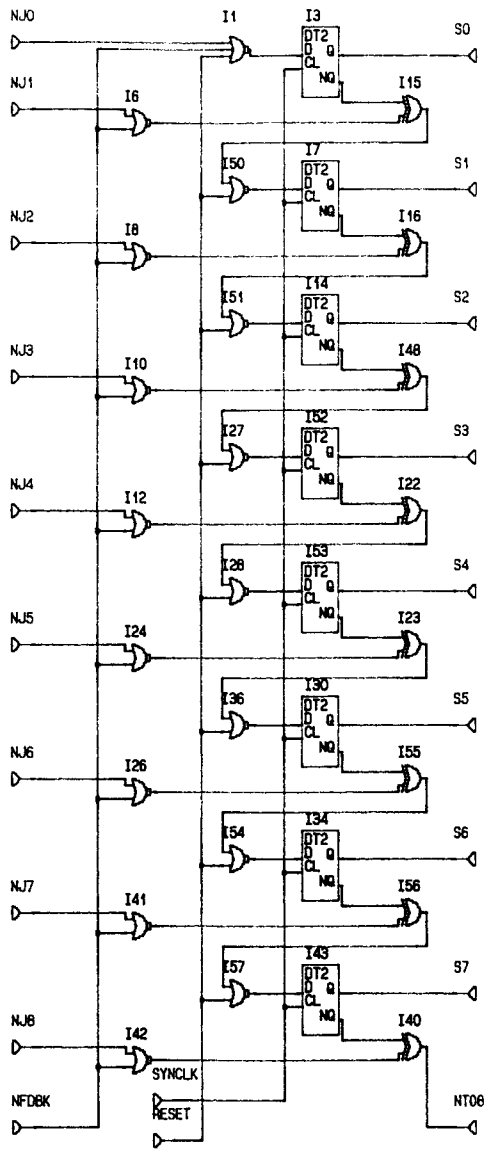
Hardcopies of the PETLD schematics were obtained directly from the design data files by using the `BXPLOT` program.

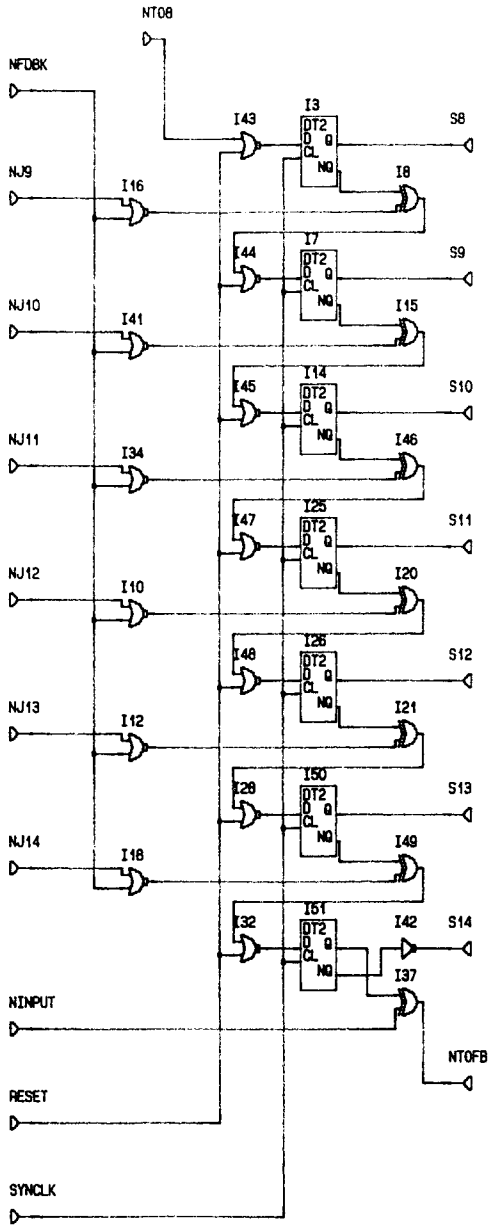

VDD
VDD
DC
+5V
GND
VSS
DC
0V
VSS
J0
J1
J2
J3
J4
J5
J6
J7
J8
J9
J10
J11
J12
J13
J14
T3
T2
T1
T0
A9
NJ0
NJ1
NJ2
NJ3
NJ4
NJ5
NJ6
NJ7
NJ8
NJ9
NJ10
NJ11
NJ12
NJ13
NJ14
NT3
NT2
NT1
NT0
NRESET
CLK
ING
INPUT
ERRATAP
ERRDET
DATOUT
NERRDET
A32
VDD
PA
GND
A31
NSINB
INPA
CLKA
NLOADB
CLKB
NS0
NS1
NS2
NS3
NS4
PIPELI
NOUTB
DATAOUT
CLK3
M0
M1
M2
M3
M4
NRESET
CLK2
A41
RESET
CLKIN
NM4
NM3
NM2
NM1
NM0
CTRLSG
ETING
NETRST
CLKOUT
ECLC
MODV
A4
NECLC
NMODV
CLK
LNDECO
ECC
CLK
A7
A30
W0
W1
W2
W3
W4

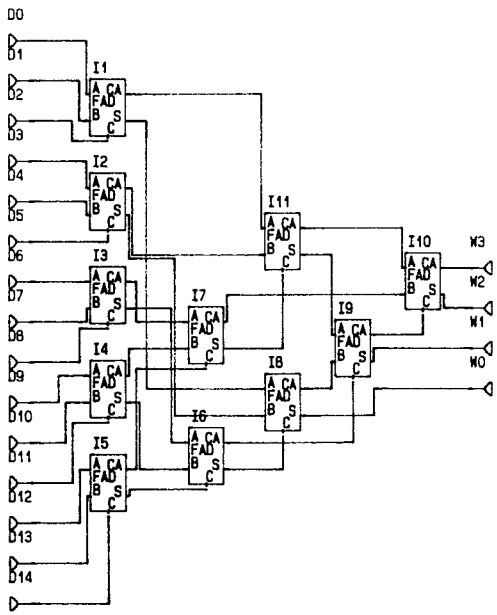

I1
NECLC
DT2
D Q
CL
NQ
I3
ECC
I2
NMODV
DT2
D Q
CL
NQ
CLK

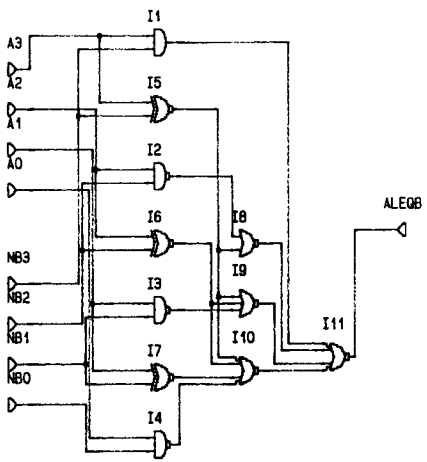

NJ0
NJ1
NJ2
NJ3
NJ4
NJ5
NJ6
NJ7
NJ8
NJ9
NJ10
NJ11
NJ12
NJ13
NJ14
NRESET
CLK
ING
INPUT
I1
I2
I3
I10
ERRDET
ALEQB
CMP4LE
WEIGHT
NERPAT
SYNCLK
OUTG
ING
INPUT
SYNREG
I24
I33
DATOUT
I5
NT3
NT2
NT1
NT0

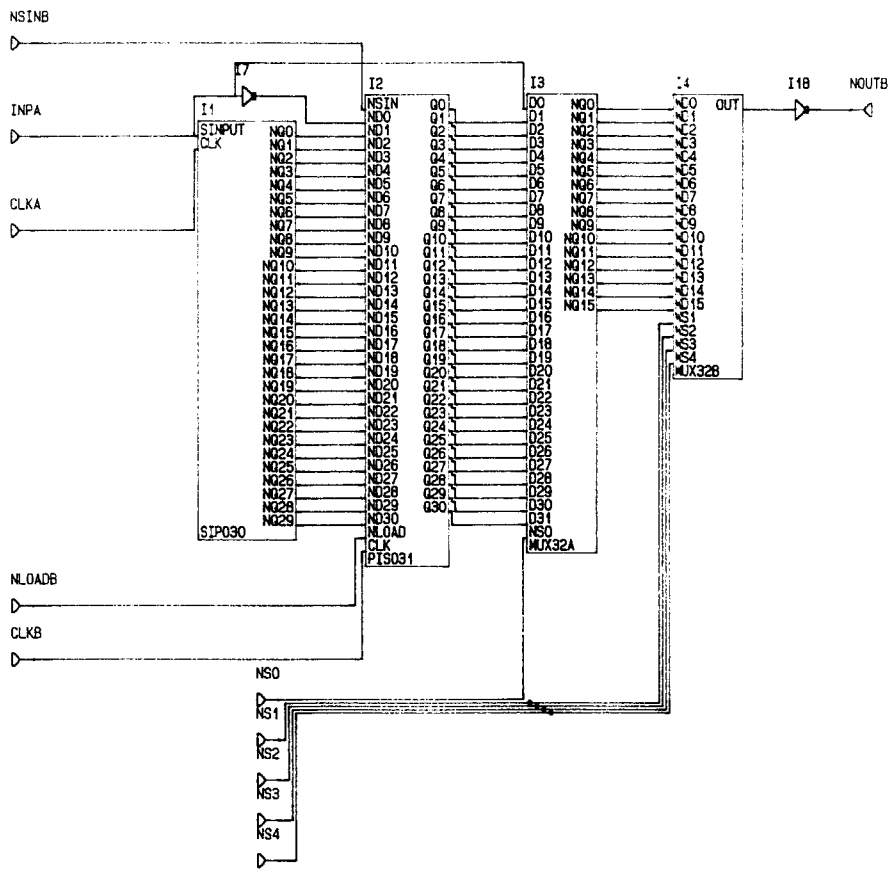

NJ0
NJ1
NJ2
NJ3
NJ4
NJ5
NJ6
NJ7
NJ8
NJ9
NJ10
NJ11
NJ12
NJ13
NJ14
I56
NJ0
NJ1
NJ2
NJ3
NJ4
NJ5
NJ6
NJ7
NJ8
SYNCLK
NFDBK
RESET
SYNRA
S0
S1
S2
S3
S4
S5
S6
S7
NTO8
I57
NTO8
NFDBK
NJ9
NJ10
NJ11
NJ12
NJ13
NJ14
NINPUT
RESET
SYNCLK
SYNRB
S8
S9
S10
S11
S12
S13
S14
NTOFB
S0
S1
S2
S3
S4
S5
S6
S7
S8
S9
S10
S11
S12
S13
S14
NRBFA
NRESET
NRBFB
I31
I44
SYNCLK
K725
CKBFA
CKBFB
I45
NERPAT
OUT6
ING
I30
INPUT

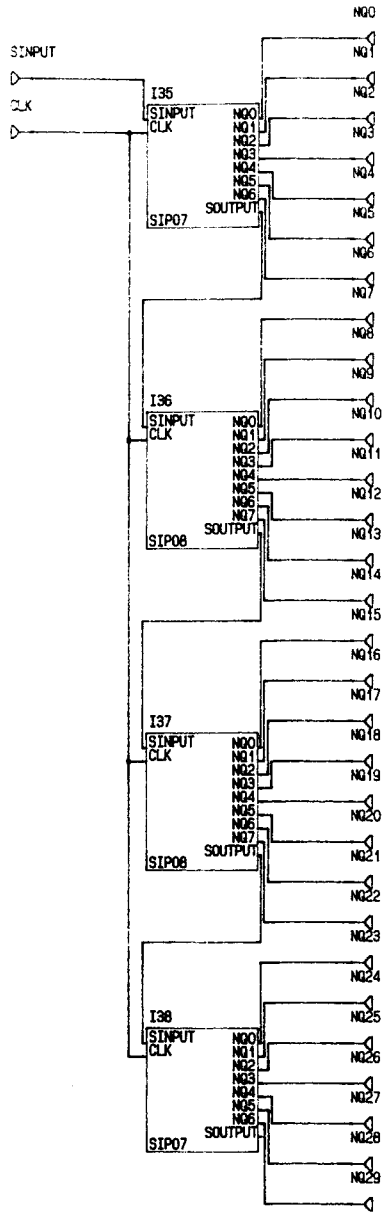

NJ0
NJ1
NJ2
NJ3
NJ4
NJ5
NJ6
NJ7
NJ8
NFDBK
SYNCLK
RESET
NT08
S0
S1
S2
S3
S4
S5
S6
S7
I1
I3
I6
I7
I8
I10
I12
I14
I15
I16
I22
I23
I24
I26
I27
I28
I30
I34
I36
I40
I41
I42
I43
I48
I50
I51
I52
I53
I54
I55
I56
I57
DT2
D
Q
CL
NQ

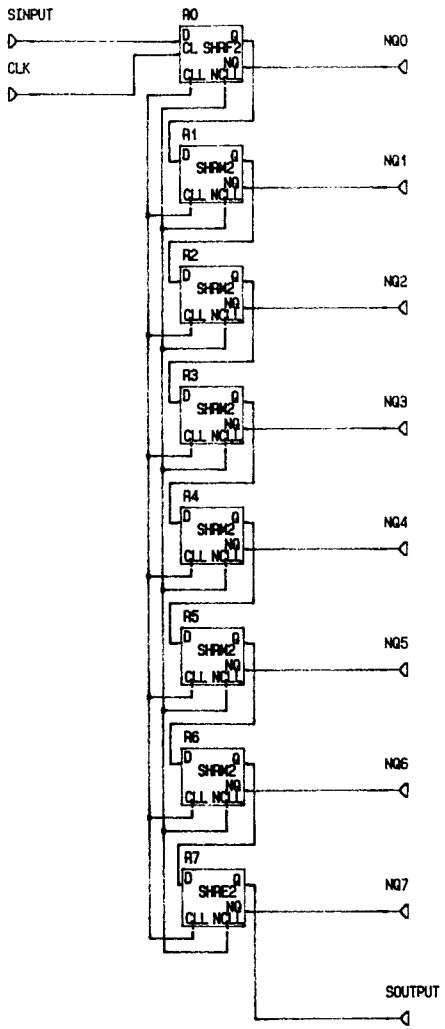

NTOB
NFDBK
NJ9
NJ10
NJ11
NJ12
NJ13
NJ14
NINPUT
RESET
SYNCLK
I43
I44
I45
I47
I48
I28
I32
I16
I41
I34
I10
I12
I18
I3
I7
I14
I25
I26
I50
I51
I8
I15
I46
I20
I21
I49
I42
I37
S8
S9
S10
S11
S12
S13
S14
NTOFB
DT2
D
Q
CL
NQ

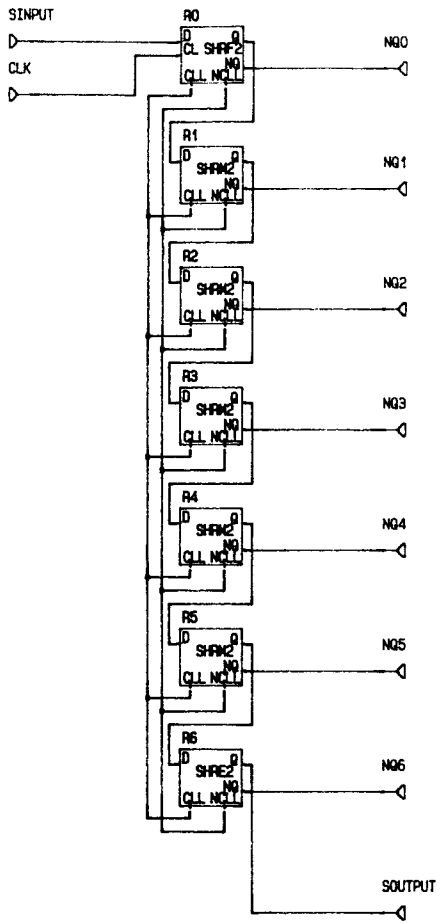

D0
D1
D2
D3
D4
D5
D6
D7
D8
D9
D10
D11
D12
D13
D14
I1
I2
I3
I4
I5
I6
I7
I8
I9
I10
I11
W3
W2
W1
W0
A
B
CA
FAD
S
C

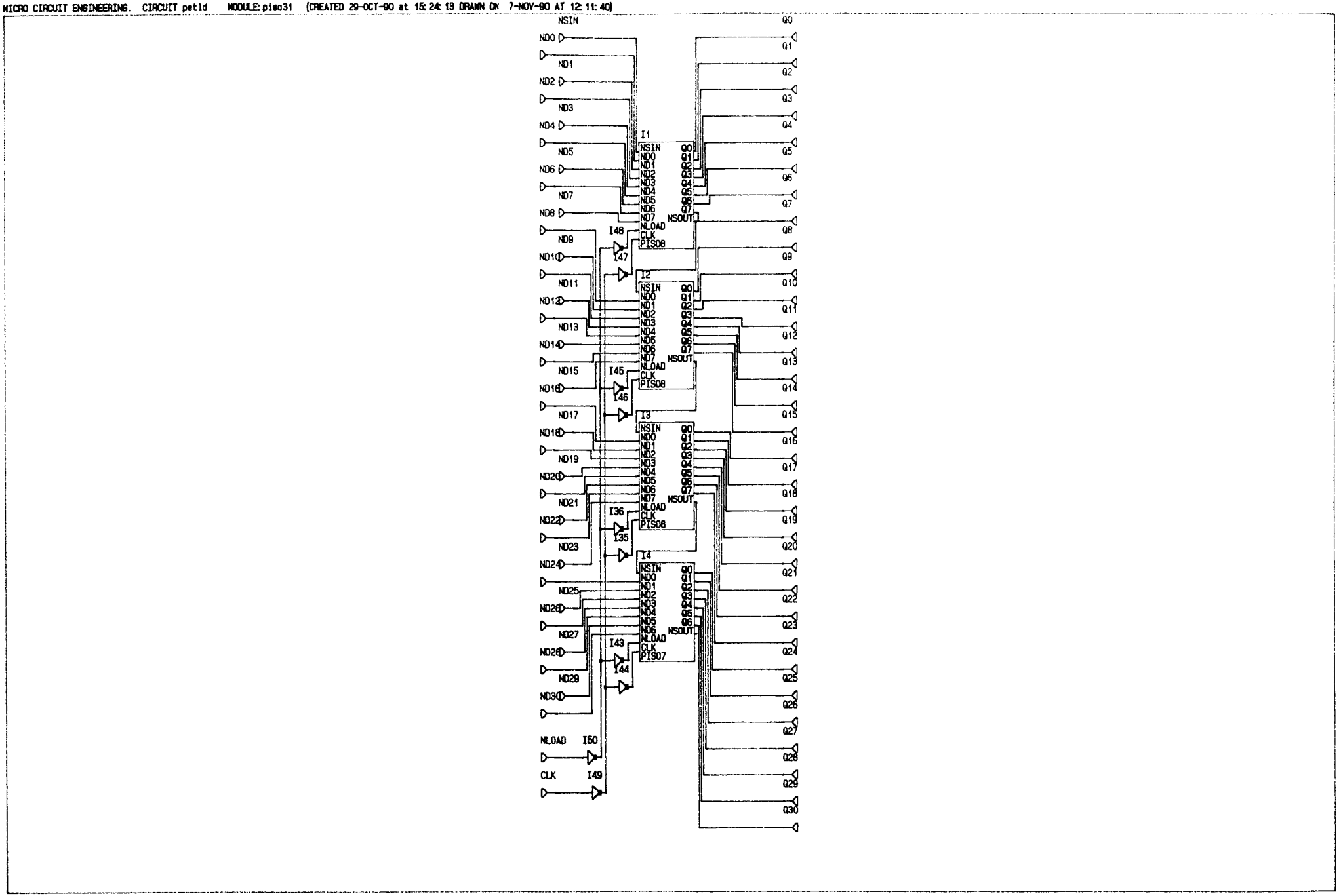

I1
A3
A2
I5
A1
A0
I2
I6
I8
ALEQB
NB3
I9
NB2
I3
I11
NB1
I10
NB0
I7
I4

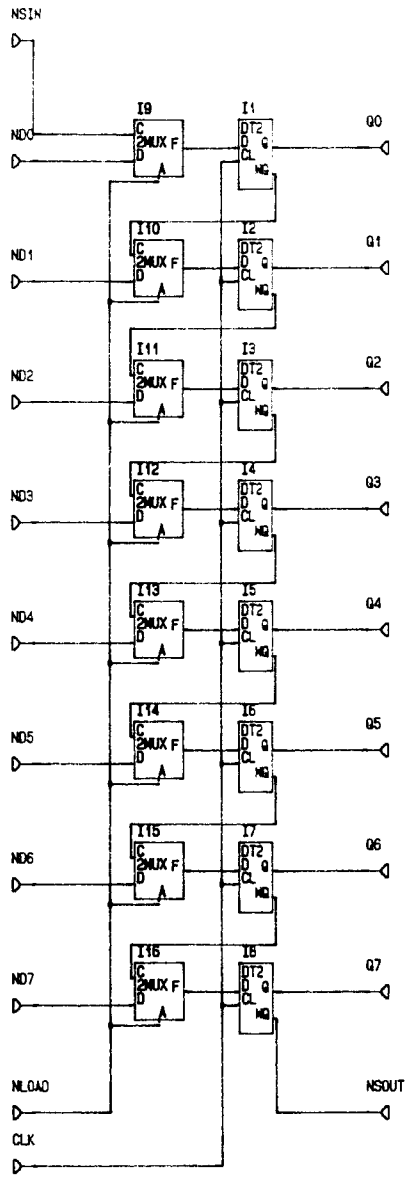

NSINB
INPA
CLKA
NLOADB
CLKB
NS0
NS1
NS2
NS3
NS4
I1
SINPUT
CLK
SIP030
I2
PIS031
I3
MUX32A
I4
MUX32B
I18
NOUTB
OUT

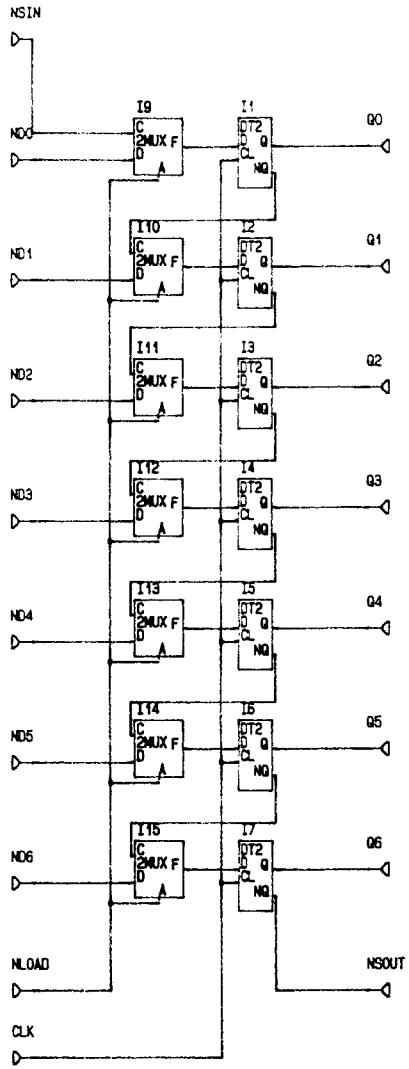

SINPUT
CLK
I35
SINPUT
CLK
NQ0
NQ1
NQ2
NQ3
NQ4
NQ5
NQ6
SOUTPUT
SIP07
I36
SINPUT
CLK
NQ0
NQ1
NQ2
NQ3
NQ4
NQ5
NQ6
NQ7
SOUTPUT
SIP08
I37
SINPUT
CLK
NQ0
NQ1
NQ2
NQ3
NQ4
NQ5
NQ6
NQ7
SOUTPUT
SIP08
I38
SINPUT
CLK
NQ0
NQ1
NQ2
NQ3
NQ4
NQ5
NQ6
SOUTPUT
SIP07
NQ0
NQ1
NQ2
NQ3
NQ4
NQ5
NQ6
NQ7
NQ8
NQ9
NQ10
NQ11
NQ12
NQ13
NQ14
NQ15
NQ16
NQ17
NQ18
NQ19
NQ20
NQ21
NQ22
NQ23
NQ24
NQ25
NQ26
NQ27
NQ28
NQ29

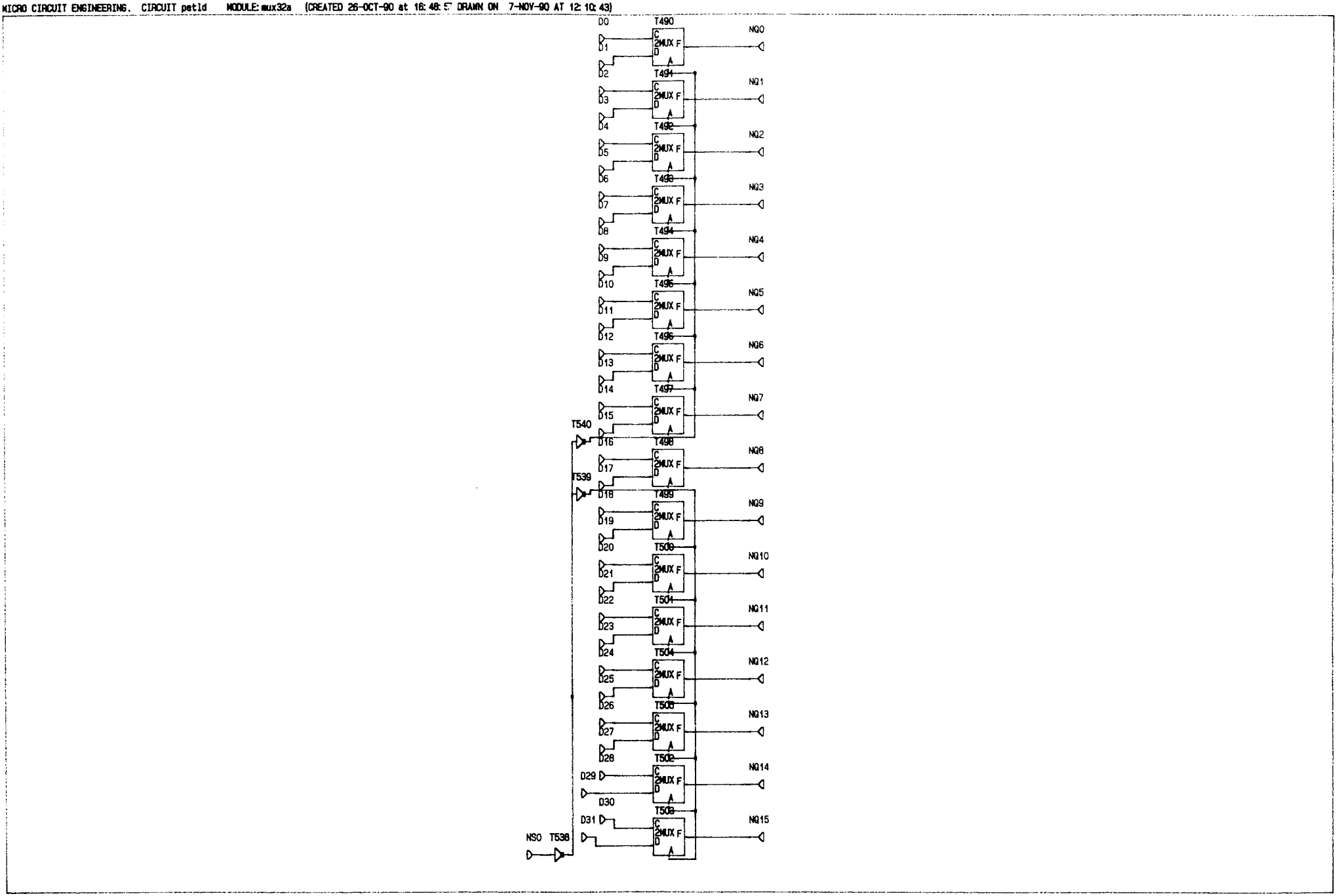

SINPUT
CLK
R0
R1
R2
R3
R4
R5
R6
R7
SHRE2
D
CL
NQ
NQ0
NQ1
NQ2
NQ3
NQ4
NQ5
NQ6
NQ7
SOUTPUT

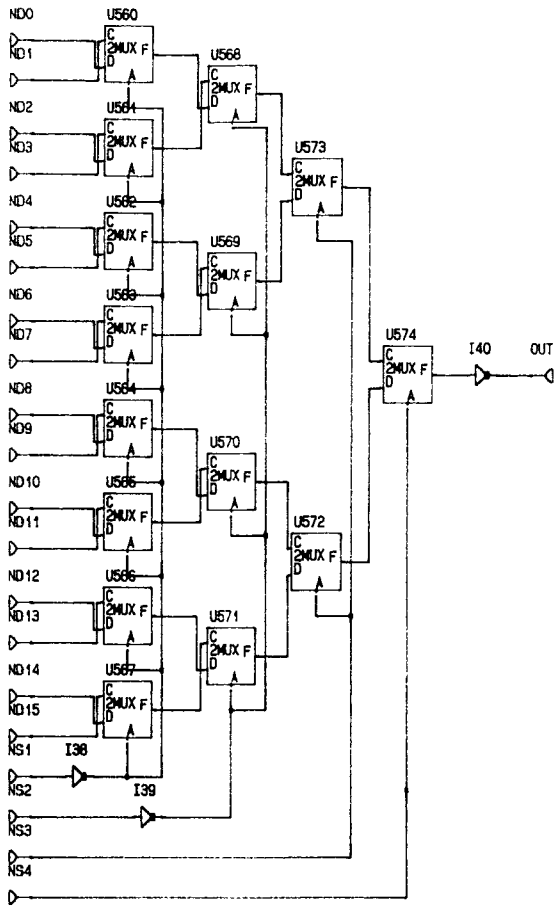

SINPUT
CLK
R0
R1
R2
R3
R4
R5
R6
SHRF2
SHRE2
D
Q
CL
NQ
NQ0
NQ1
NQ2
NQ3
NQ4
NQ5
NQ6
SOUTPUT

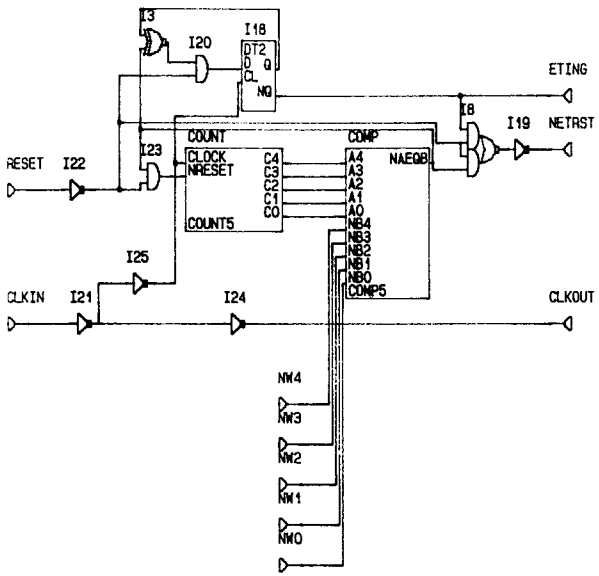

NSIN
ND0
ND1
ND2
ND3
ND4
ND5
ND6
ND7
ND8
ND9
ND10
ND11
ND12
ND13
ND14
ND15
ND16
ND17
ND18
ND19
ND20
ND21
ND22
ND23
ND24
ND25
ND26
ND27
ND28
ND29
ND30
NLOAD
CLK
I1
I2
I3
I4
I47
I48
I45
I46
I36
I35
I43
I44
I49
I50
PIS08
PIS07
NSOUT
Q0
Q1
Q2
Q3
Q4
Q5
Q6
Q7
Q8
Q9
Q10
Q11
Q12
Q13
Q14
Q15
Q16
Q17
Q18
Q19
Q20
Q21
Q22
Q23
Q24
Q25
Q26
Q27
Q28
Q29
Q30

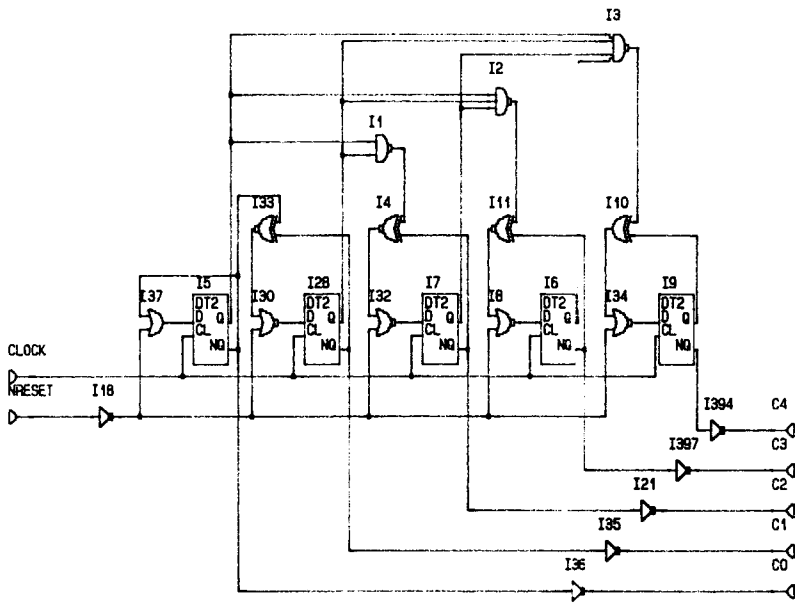

NSIN
I9
I1
ND0
Q0
I10
I2
ND1
Q1
I11
I3
ND2
Q2
I12
I4
ND3
Q3
I13
I5
ND4
Q4
I14
I6
ND5
Q5
I15
I7
ND6
Q6
I16
I8
ND7
Q7
NLOAD
NSOUT
CLK
C
2MUX F
D
A
DT2
D Q
CL
NQ

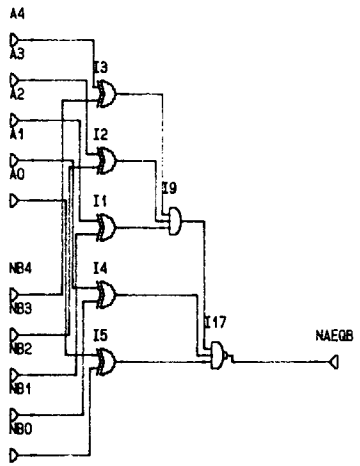

NSIN
I9
I1
ND
MUX F
DT2
Q0
NQ
I10
I2
ND1
Q1
I11
I3
ND2
Q2
I12
I4
ND3
Q3
I13
I5
ND4
Q4
I14
I6
ND5
Q5
I15
I7
ND6
Q6
NLOAD
NSOUT
CLK


D0
D1
D2
D3
D4
D5
D6
D7
D8
D9
D10
D11
D12
D13
D14
D15
D16
D17
D18
D19
D20
D21
D22
D23
D24
D25
D26
D27
D28
D29
D30
D31
T490
T491
T492
T493
T494
T495
T496
T497
T498
T499
T500
T501
T504
T505
T502
T503
T540
T539
NS0
T538
NQ0
NQ1
NQ2
NQ3
NQ4
NQ5
NQ6
NQ7
NQ8
NQ9
NQ10
NQ11
NQ12
NQ13
NQ14
NQ15
C
2MUX F
D
A


ND0
ND1
ND2
ND3
ND4
ND5
ND6
ND7
ND8
ND9
ND10
ND11
ND12
ND13
ND14
ND15
NS1
NS2
NS3
NS4
U560
U561
U562
U563
U564
U565
U566
U567
U568
U569
U570
U571
U572
U573
U574
I38
I39
I40
OUT
C
2MUX F
D
A


I3
I20
I18
DT2
D Q
CP
NQ
ETING
I8
I19
NETRST
COUNT
CLOCK
NRESET
COUNT5
C4
C3
C2
C1
C0
COMP
A4
A3
A2
A1
A0
NB4
NB3
NB2
NB1
NB0
COMP5
NAEQB
I23
RESET
I22
I25
CLKIN
I21
I24
CLKOUT
NW4
NW3
NW2
NW1
NW0


I3
I2
I1
I33
I4
I11
I10
I5
I28
I7
I6
I9
I37
I30
I32
I8
I34
DT2
D Q
CL NQ
CLOCK
NRESET
I18
I394
C4
I397
C3
I21
C2
I35
C1
I36
C0


A4
A3
A2
A1
A0
NB4
NB3
NB2
NB1
NB0
I3
I2
I1
I4
I5
I9
I17
NAEQB

Appendix B

Annotated Test Program

```
*WAVE PETLD

! ************************************************************
! Test Pattern for PETLD  -  C. Fonseca
!
! Group 1 - Hamming (7,4)
!
!    1st vector (unmodified) = 1010011  decodes to 0011;
!    2nd vector (unmodified) = 1000101  decodes to 0101,
!    3rd vector (unmodified) = 0101010  decodes to 1010;
!                                ^error in check bit...
!    4th vector (unmodified) = 0101101  decodes to 1100;
!                                     ^error in message bit...
!    5th vector ( modified ) = 0101010,
!        modification vector = 1010101,
!          unmodified vector = 1111111  decodes to 1111;
!
! ************************************************************
!
! Groups 2 & 3 - BCH (31,16)
```

```
!
!    1st vector (unmodified) = 00111010111001001101000000000000
!        decodes to 0110100000000000
!
!    2nd vector (unmodified) = 00111010111001001101001000010000
!                                              2 errors ^    ^
!        decodes to 0110100000000000 (errors corrected)
!
!    3rd vector (unmodified) = 00110010101001001111000000000000
!                                    ^    ^        ^ 3 errors
!        decodes to 0110100000000000 (errors corrected)
!
!    4th vector (unmodified) = 00111010101001001101100000000001
!                                3 errors ^         ^         ^
!        decodes to 0110110000000001 (errors not corrected)
!
!    5th vector ( modified ) = 01010101010101010101010101010101
!        modification vector = 10101010101010101010101010101010
!          unmodified vector = 11111111111111111111111111111111
!        decodes to 1111111111111111.
!
!    6th vector (unmodified) = 11110110101110101110101111111101
!                                  ^ ^ 3 errors (end-around)  ^
!        decodes to 0111010111111100 (errors corrected)
!
! ***************************************************************
!
! Groups 4 & 5 - Input & Output registers checking
!
! ***************************************************************
```

```
!
! Groups 6 & 7 - Syndrome register checking
!
! ************************************************************

! DEFINE TIMING

PERIOD 1000
TG0 0 500
TG1 300 500
TG2 300 500
TG3 300 500
TG4 980 10

*TYPES

J0 I TG0
J1 I TG0
J2 I TG0
J3 I TG0
J4 I TG0
J5 I TG0
J6 I TG0
J7 I TG0
J8 I TG0
J9 I TG0
J10 I TG0
J11 I TG0
J12 I TG0
J13 I TG0
```

```
J14 I TG0
W0 I TG0
W1 I TG0
W2 I TG0
W3 I TG0
W4 I TG0
M0 I TG0
M1 I TG0
M2 I TG0
M3 I TG0
M4 I TG0
T0 I TG0
T1 I TG0
T2 I TG0
T3 I TG0
NRESET I TG0
ECLC I TG0
MODV I TG0
DATAOUT O TG4
NERRDET O TG4
CLK1 I TG1
CLK2 N TG2
CLK3 I TG3

*PATTERNS

/ ! GROUP 1 - Hamming (7,4)

@0:
!Clock Signals...
```

```
   CLK1=1(&1 A1,1 TIL &99),CLK2=1,CLK3=1(&1 A1,2,2,2 TIL &99),

!Shifted Generator Polynomial (x^3 + x + 1) ...
   J0=0,J1=0,J2=0,J3=0,J4=0,J5=0,J6=0,J7=0,J8=0,J9=0,
   J10=0,J11=0,J12=1,J13=1,J14=0,

!Codeword length (W=7) ...
   W0=1,W1=1,W2=1,W3=0,W4=0,

!No. of message bits (M=4) ...
   M0=0,M1=0,M2=1,M3=0,M4=0,

!Error-correcting capability (T=1) ...
   T0=1,T1=0,T2=0,T3=0,

!Frame signal...
   NRESET=1(&1 A1,27 TIL &99),

!Input signal...
@0:ECLC=1(&2 0,&2 1,&2 0,&4 1,&4   1,&2 0,&6 1,&2 0,&2 1,&2
        0,&2 1,&2 0,&2 1,&2 0,&2 1,&2 0,&2   0,&2 1,&2 0,&2 1,&4 0,&2 1,&2
        0,&2 1,&2 0,&2 1,&2 0,&2 1,&2 0,&2   0),

! Modification vector...
@0:MODV=0(&56 1,&2 0,&2 1,&2 0,&2 1,&2 0,&2 1,&2   0),

!Output signal...
@0:DATAOUT=X(&29   0,&3 0,&4 1,&3 1,&4   0,&3 1,&4 0,&3 1,&4
                   1,&3 0,&4 1,&3 0,&4   1,&3 1,&4 0,&3 0,&4
                   1,&3 1,&4 1,&3 1,&4   0),
```

```
   NERRDET=X(&22 1,&7 X,&7 1,&7 X,&7 1,&7 X,&7 0,&4 1,&3 X,&7 1),


/ ! GROUP 2 - BCH (31,16)

@0:
!Clock Signals...
   CLK1=1(&1 A1,1 TIL &247),CLK2=1,
   CLK3=1(&1 A1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2 TIL &247),

!Shifted Generator Polynomial ( ... ) ...
   J0=1,J1=1,J2=1,J3=1,J4=0,J5=1,J6=0,J7=1,J8=1,J9=1,
   J10=1,J11=1,J12=0,J13=0,J14=0,

!Codeword length (W=31) ...
   W0=1,W1=1,W2=1,W3=1,W4=1,

!No. of message bits (M=16) ...
   M0=0,M1=0,M2=0,M3=0,M4=1,

!Error-correcting capability (T=3) ...
   T0=1,T1=1,T2=0,T3=0,

!Frame signal...
   NRESET=1(&1 A1,61 TIL &247),

!Input signal...
   ECLC=0(&4 1,&6 0,&2 1,&2 0,&2 1,&6 0,&4 1,&2 0,&4 1,&4 0,&2 1,&2 0,&22
        0,&4 1,&6 0,&2 1,&2 0,&2 1,&6 0,&4 1,&2 0,&4 1,&4 0,&2 1,&2
        0,&4 1,&2 0,&8 1,&2 0,&6
```

```
        0,&4 1,&4 0,&4 1,&2 0,&2 1,&2 0,&2 1,&2 0,&4 1,&2 0,&4 1,&8 0,&22
        0,&4 1,&6 0,&2 1,&2 0,&2 1,&2 0,&2 1,&2 0,&4 1,&2 0,&4 1,&4
        0,&2 1,&4 0,&18 1    ),

! Modification vector...
   MODV=0,

!Output signals...
   DATAOUT=X(&125 0,&3 1,&8 0,&4 1,&4 0,&43
                  0,&3 1,&8 0,&4 1,&4 0     ),
   NERRDET=X(&94 1,&31 X,&31 0,&13 1,&18 X,&31 0,&4 1 ),

/ ! GROUP 3 - BCH (31,16) (cont.)

@0:
!Clock Signals...
   CLK1=1(&1 A1,1 TIL &251),CLK2=1,
   CLK3=1(&1 A1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2 TIL &251),

!Shifted Generator Polynomial ( ... ) ...
   J0=1,J1=1,J2=1,J3=1,J4=0,J5=1,J6=0,J7=1,J8=1,J9=1,
   J10=1,J11=1,J12=0,J13=0,J14=0,

!Codeword length (W=31) ...
   W0=1,W1=1,W2=1,W3=1,W4=1,

!No. of message bits (M=16) ...
   M0=0,M1=0,M2=0,M3=0,M4=1,

!Error-correcting capability (T=3) ...
```

```
   T0=1,T1=1,T2=0,T3=0,

!Frame signal...
   NRESET=1(&1 A1,61 TIL &251),

!Input signal...
   ECLC=0(&2 1,&2 0,&2 1,&2 0,&2 1,&2 0,&2 1,&2 0,&2 1,&2 0,&2 1,&2 0,&2
        1,&2 0,&2 1,&2 0,&2 1,&2 0,&2 1,&2 0,&2 1,&2 0,&2 1,&2 0,&2 1,&2
        0,&2 1,&2 0,&2 1,&2 0,&2
        1,&8 0,&2 1,&4 0,&2 1,&2 0,&2 1,&6 0,&2 1,&2 0,&2 1,&6 0,&2 1,&2
        0,&2 1,&14 0,&2 1,&2  0),

! Modification vector...
   MODV=1(&2 0,&2 1,&2 0,&2 1,&2 0,&2 1,&2 0,&2 1,&2 0,&2 1,&2 0,&2 1,&2
        0,&2 1,&2 0,&2 1,&2 0,&2 1,&2 0,&2 1,&2 0,&2 1,&2 0,&2 1,&2 0,&2
        1,&2 0,&2 1,&2 0,&2 1,&2   0),

!Output signals...
   DATAOUT=0(&1 0,&3 1,&8 0,&4 1,&4 0,&43
                0,&3 1,&8 0,&4 1,&8 0,&35 1,&4
                1,&62
                0,&3 1,&12 0,&4 1,&4 0,&4 1,&27 0    ),
   NERRDET=1(&1 X,&31 0,&31 X,&31 1,&31 X,&31 0,&22 1),


/ ! Group 4 - Input and output registers checking...
  !           (no check bits...)

@0:
!Clock Signals...
```

```
   CLK1=1(&1 A1,1 TIL &125 &62 A1,1 TIL @248),
   CLK2=1(&126 0,&62 1),
   CLK3=1(&1 A1,1 TIL &125 &62 A1,1 TIL @248),

!Shifted Generator Polynomial (1) ...
   J0=0,J1=0,J2=0,J3=0,J4=0,J5=0,J6=0,J7=0,J8=0,J9=0,
   J10=0,J11=0,J12=0,J13=0,J14=0,

!Codeword length (W=31) ...
   W0=1(&157 A1,1 TIL &32),
   W1=1(&158 A2,2 TIL &31),
   W2=1(&160 A4,4 TIL &29),
   W3=1(&164 A8,8 TIL &25),
   W4=1(&172 A16,16 TIL &17),

!No. of message bits (M=31) ...
   M0=1(&126 A1,1 TIL &31,&31 1),
   M1=1(&127 A2,2 TIL &30,&31 1),
   M2=1(&129 A4,4 TIL &28,&31 1),
   M3=1(&133 A8,8 TIL &24,&31 1),
   M4=1(&141 A16,16 TIL &16,&31 1),

!Error-correcting capability (T=0) ...
   T0=0,T1=0,T2=0,T3=0,

!Frame signal...
   NRESET=1(&1P1),

!Input signal...
@0:ECLC=1(&2 A2,2 TIL @126 &62 A2,2 TIL @248),
```

```
! Modification vector...
@0:MODV=0,

! Output signals...
@0:DATAOUT=X(&125 1,&1 A1,1 TIL &62 &1 A2,2 TIL @248),
   NERRDET=X(&1 1),

/ ! Group 5 - Input and output registers checking (cont.) ...

@0:
!Clock Signals...
   CLK1=1(&1 A1,1 TIL &1 &62 A1,1 TIL @126),
   CLK2=1(&2 0,&62 1),
   CLK3=1(&1 A1,1 TIL &1 &62 A1,1 TIL @126),

!Shifted Generator Polynomial (1) ...
   J0=0,J1=0,J2=0,J3=0,J4=0,J5=0,J6=0,J7=0,J8=0,J9=0,
   J10=0,J11=0,J12=0,J13=0,J14=0,

!Codeword length (W=31) ...
   W0=1(&33 A1,1 TIL &32),
   W1=1(&34 A2,2 TIL &31),
   W2=1(&36 A4,4 TIL &29),
   W3=1(&40 A8,8 TIL &25),
   W4=1(&48 A16,16 TIL &17),

!No. of message bits (M=31) ...
   M0=1(&2 A1,1 TIL &31,&31 1),
   M1=1(&3 A2,2 TIL &30,&31 1),
```

```
   M2=1(&5 A4,4 TIL &28,&31 1),
   M3=1(&9 A8,8 TIL &24,&31 1),
   M4=1(&17 A16,16 TIL &16,&31 1),

!Error-correcting capability (T=0) ...
   T0=0,T1=0,T2=0,T3=0,

!Frame signal...
   NRESET=1(&1P1),

!Input signal...
@0:ECLC=0(&2 0,&62 A2,2 TIL @126),

! Modification vector...
@0:MODV=0,

!Output signal...
@0:DATAOUT=1(&1 0,&1 A1,1 TIL &62 &1 A2,2 TIL @126),
   NERRDET=1,


/ ! GROUP 6 - Syndrome register checking ...

@0:
!Clock Signals...
   CLK1=1(&1 A1,1 TIL &247),CLK2=1,
   CLK3=1(&1 A1,1 TIL &247),

!Shifted Generator Polynomial ( ... ) ...
   J0=1(&126 0,&62 0),
```

```
   J1=0(&126 1,&62 0),
   J2=0(&126 0,&62 1),
   J3=1(&126 0,&62 0),
   J4=0(&126 1,&62 0),
   J5=0(&126 0,&62 1),
   J6=1(&126 0,&62 0),
   J7=0(&126 1,&62 0),
   J8=0(&126 0,&62 1),
   J9=1(&126 0,&62 0),
   J10=0(&126 1,&62 0),
   J11=0(&126 0,&62 1),
   J12=1(&126 0,&62 0),
   J13=0(&126 1,&62 0),
   J14=0(&126 0,&62 1),

!Codeword length (W=31) ...
   W0=1,W1=1,W2=1,W3=1,W4=1,

!No. of message bits (M=31) ...
   M0=1,M1=1,M2=1,M3=1,M4=1,

!Error-correcting capability (T=15) ...
   T0=1,T1=1,T2=1,T3=1,

!Frame signal...
   NRESET=1(&1 A1,61 TIL &247),

!Input signal...
   ECLC=0(&60 A2,60 TIL &186),
```

```
! Modification vector...
   MODV=0,

!Output signals...
   DATAOUT=X(&125 0,&4P2,&4P2,&4P2,&4P2,&4P2,&30P2,
                    &2P2,&4P2,&4P2,&4P2,&4P2,&32 ),
   NERRDET=1,

/ ! GROUP 7 - Syndrome register checking (cont.) ...

@0:
!Clock Signals...
   CLK1=1(&1 A1,1 TIL &65),CLK2=1,
   CLK3=1(&1 A1,1 TIL &65),

!Shifted Generator Polynomial ( ... ) ...
   J0=0,J1=0,J2=0,J3=0,J4=0,J5=0,J6=0,J7=0,J8=0,J9=0,
   J10=0,J11=0,J12=0,J13=0,J14=0,

!Codeword length (W=31) ...
   W0=1,W1=1,W2=1,W3=1,W4=1,

!No. of message bits (M=31) ...
   M0=1,M1=1,M2=1,M3=1,M4=1,

!Error-correcting capability (T=15) ...
   T0=1,T1=1,T2=1,T3=1,

!Frame signal...
   NRESET=1(&1 A1,61 TIL &65),
```

```
!Input signal...
   ECLC=0,

! Modification vector...
   MODV=0,

!Output signals...
   DATAOUT=1(&3 0,&4P2,&4P2,&4P2,&4P2,&34P2),
   NERRDET=1,

*END
```

Appendix C

PETLD Data Sheet (MT76432)

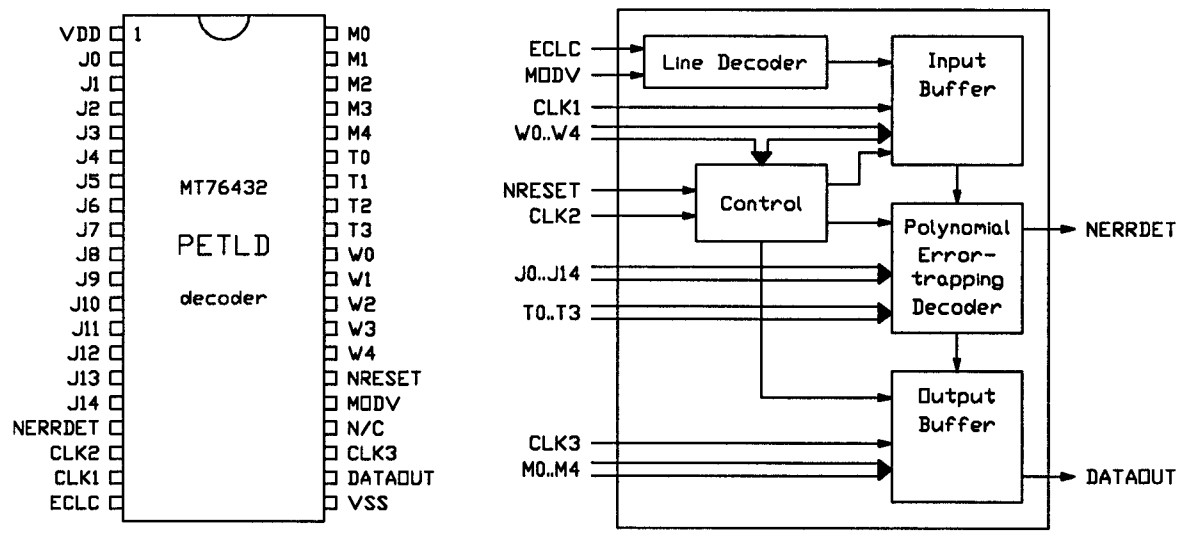


Figure C.1: Device pin out and block diagram

C.1 Description

This device is a flexible runlength limited error control code (RLECC) decoder and provides both error detection and correction facilities. It will handle any cyclic error control code, within the following bounds:

- maximum code word length $N_{\max} = 31$;
- maximum message length $K_{\max} = 31$;
- maximum number of check bits $Q_{\max} = 15$.

Line decoding is performed by the mod-2 addition of an externally supplied Modification Vector. This recovers the original (unmodified) error-control code words (ECCs), which are decoded using error-trapping.

Code word and modification vector bits are supplied at the ECLC and MODV pins, respectively. Unmodified code words are then fed into a polynomial decoder. The code to be handled is selected by pins J0-J14. Pins T0-T3 specify the random error correction capability of the code and the code word and message lengths are specified by pins W0-W4 and M0-M4 respectively.

Code word and modification vector bits are clocked into the device on negative edges of CLK1 and message bits are clocked out of the DATAOUT pin on negative edges of CLK3. The NERRDET signal, which contains information about error detection and correction, is clocked out on negative edges of CLK2, an auxiliary clock used by the polynomial decoder. Clock frequency relationships should be as follows:

$$\begin{aligned} f_{\text{CLK2}} &= 2\, f_{\text{CLK1}} \\ f_{\text{CLK3}} &= \frac{k}{n} f_{\text{CLK1}} \end{aligned}$$

The NRESET pin is used to initialise the device. It needs to be held low once, as a reset, until after the falling edge of CLK2 (see Figure C.2) to which the other signals relate. Alternatively, a "frame" signal may also be applied, as shown in Figure C.3.

C.2 DC Characteristics

Parameter	Test Conditions	Min	Typ	Max	Unit
V_{DD}		4.5	5.0	5.5	V
I_{DD} (static)		—	—	1.5	mA
V_{IH}		2.4	—	$V_{DD} + 0.3$	V
V_{IL}		−0.3	—	0.8	V
I_{IN}	$V_{IH} = V_{DD}$	−5	—	5	μA
	$V_{IL} = 0$ V	−5	—	5	μA
V_{OH}	$I_{OH} = 1$ mA	2.8	—	—	V
V_{OL}	$I_{OL} = 3.2$ mA	—	—	0.4	V

Operating Ambient Temperature Range: 0 to 70°C

C.3 AC Characteristics

Pin	Param	Min	Typ	Max	Unit	Comments
ECLC	t_{setup}	28	—	—	ns	CLK1
	t_{hold}	43	—	—	ns	CLK1
MODV	t_{setup}	28	—	—	ns	CLK1
	t_{hold}	43	—	—	ns	CLK1
J0-J14	t_{setup}	55	—	—	ns	CLK2
	t_{hold}	46	—	—	ns	CLK2
NRESET	t_{setup}	91	—	—	ns	CLK2
	t_{hold}	36	—	—	ns	CLK2

Pin	Param	Min	Typ	Max	Unit	Comments
T0-T3	t_{setup}	257	—	—	ns	CLK2
	t_{hold}	−1	—	—	ns	CLK2
	t_{setup}	250	—	—	ns	CLK3
	t_{hold}	−9	—	—	ns	CLK3
W0-W4	t_{setup}	329	—	—	ns	CLK2
	t_{hold}	23	—	—	ns	CLK2
	t_{setup}	257	—	—	ns	CLK3
	t_{hold}	−3	—	—	ns	CLK3
DATAOUT	t_{delay}	—	168	286	ns	CLK3
	t_{delay}	—	150	255	ns	M0-M4
	t_{delay}	—	266	453	ns	W0-W4[a]
NERRDET	t_{delay}	—	384	653	ns	CLK2
	t_{delay}	—	122	208	ns	T0-T3
CLK2	T	703	—	—	ns	
CLK1	t_1	0	—	—	ns	see Fig. C.2
	t_2	118	—	—	ns	
CLK3	t_3	−5	—	—	ns	see Fig. C.2
	t_4	686	—	—	ns	

[a]for Mi = 0, CLKi = off, test only.

Notes: 1. All parameters specified about the 1.5 V point.

2. Delay times (t_{delay}) assume 100 pF loading.

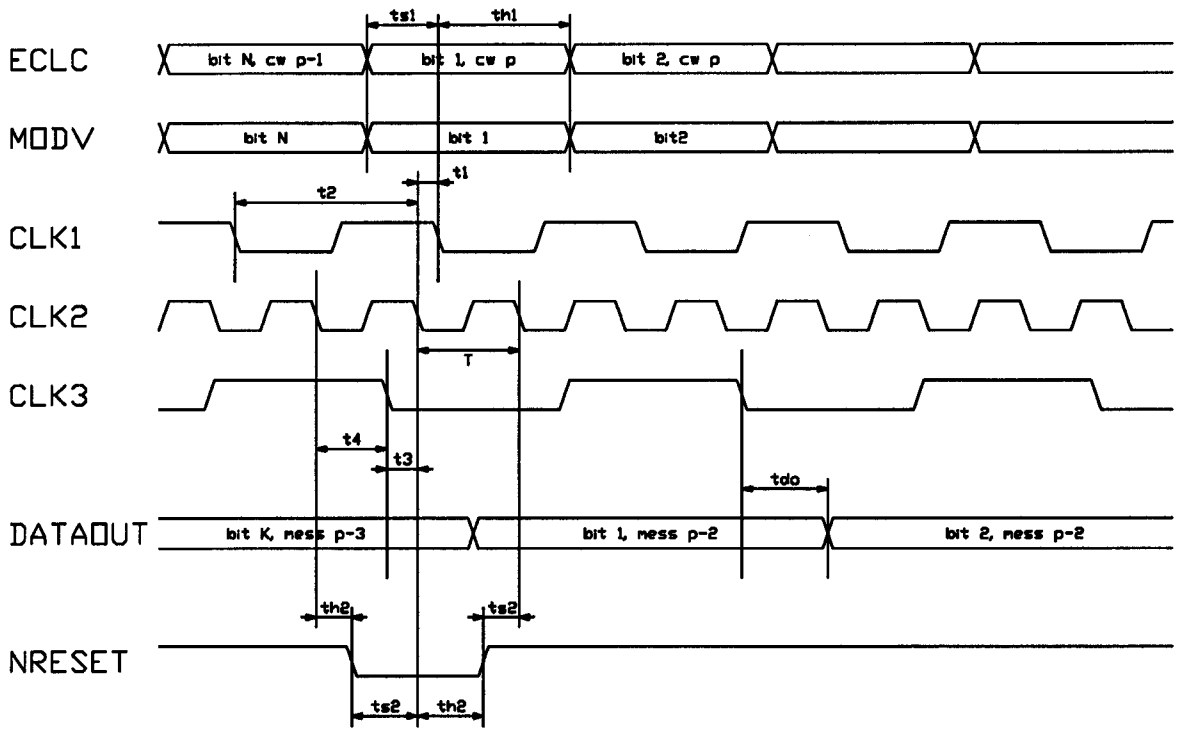
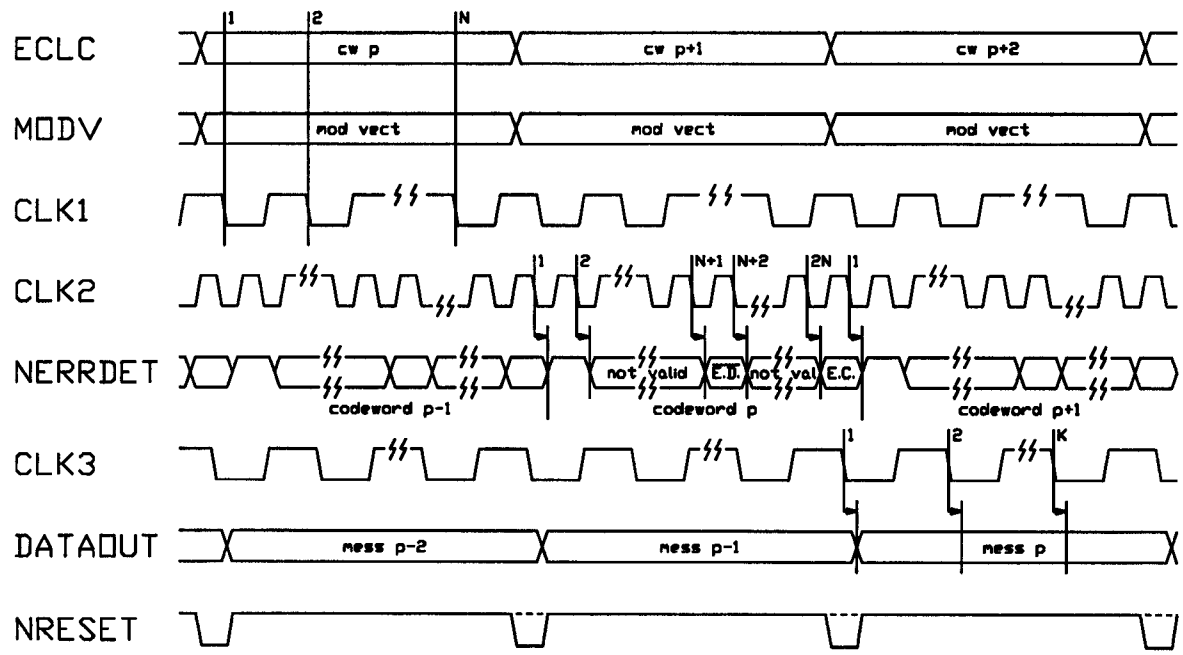


Figure C.2: System timing (bit level)



Note: $\overline{\text{E.D.}}$ — errors detected in message bits (active low);
E.C. — errors corrected (active high).

Figure C.3: System timing (frame level)

C.4 Operating example

Code: BCH(15,5)

- code word length $n = 15$;
- message length $k = 5$;
- number of check bits $q = n - k = 10$;
- random error correcting capability $t = 3$;
- generator polynomial $\mathbf{g}(X) = X^{10} + X^8 + X^5 + X^4 + X^2 + X + 1$.

The code is specified by a word called **J**, derived from $\mathbf{g}(X)$ in the following manner:

1. Take the binary representation of $\mathbf{g}(X)$, i.e., 10100110111.
2. Drop the leading "1". This gives 0100110111.
3. Add $15-q$ zeros after the rightmost bit to obtain a 15-bit word, i.e., 010011011100000.

The added zeros correspond to those elements in the syndrome register that will not be used. J pins should be set according to the word obtained, as follows:

J14	J13	J12	J11	J10	J9	J8	J7	J6	J5	J4	J3	J2	J1	J0
0	1	0	0	1	1	0	1	1	1	0	0	0	0	0

The rest of the configuration pins give information about word length, message length and error-correction capability, and should be set in the following way:

W4	W3	W2	W1	W0	
0	1	1	1	1	$n = 15$;

M4	M3	M2	M1	M0	
0	0	1	0	1	$k = 5$;

T3	T2	T1	T0	
0	0	1	1	$t = 3$.

Appendix D

Test Board Design

This appendix contains specific information relative to the test circuit discussed in chapter 5.

D.1 Schematics

The circuit schematics are given in figures D.1 to D.6. In figure D.1 the circuit is split into five modules, represented as boxes, together with their inputs and outputs. The remaining figures provide schematic description of each module.

D.2 EPROM contents

The contents of the EPROM used in figure D.5 to generate the control signals for the circuit are given in table D.1. The corresponding timing diagram is shown in figure D.7.

D.3 Parts list

The list of components used by the design is given at the end of this appendix.

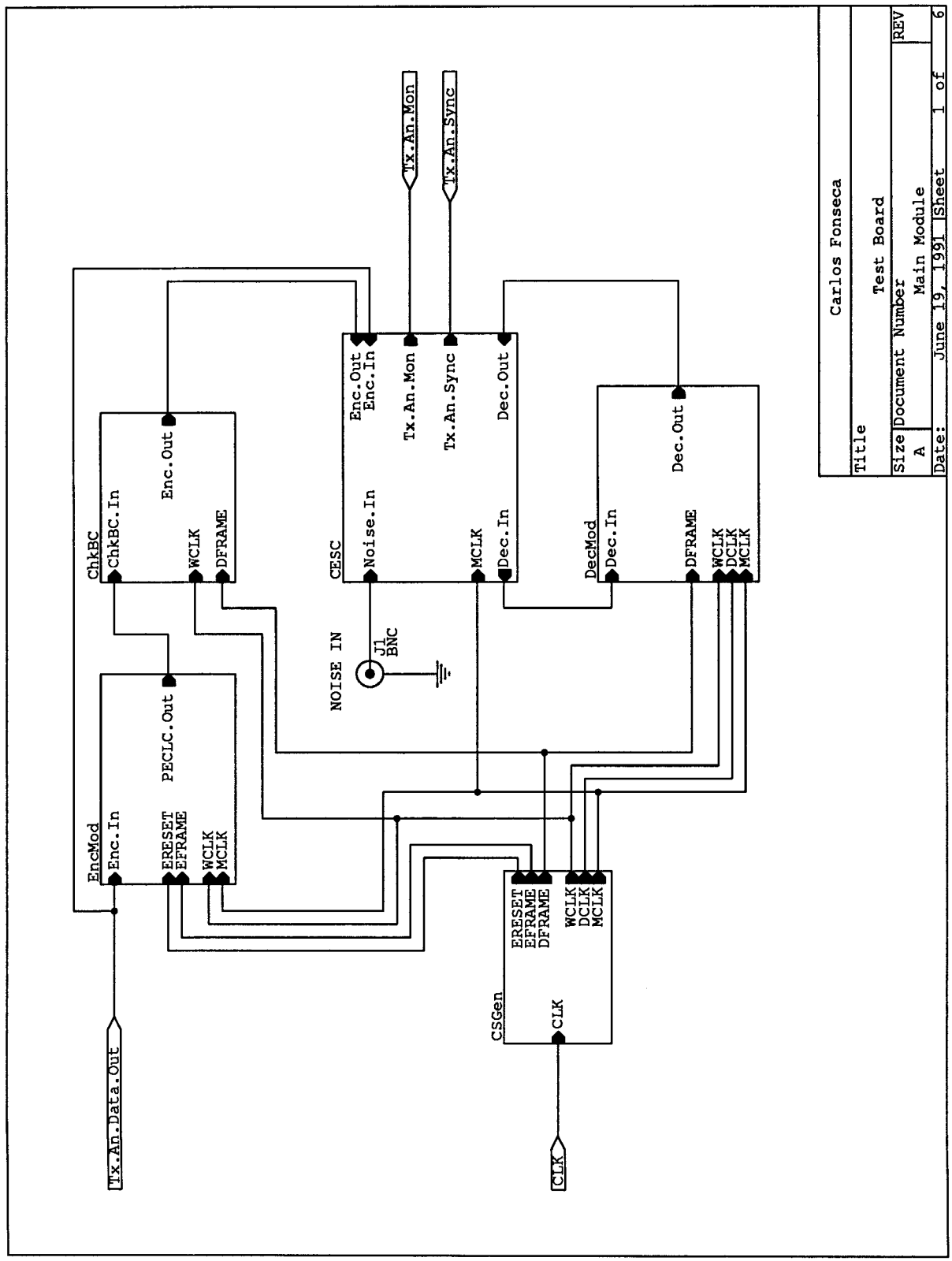


Figure D.1: Main module

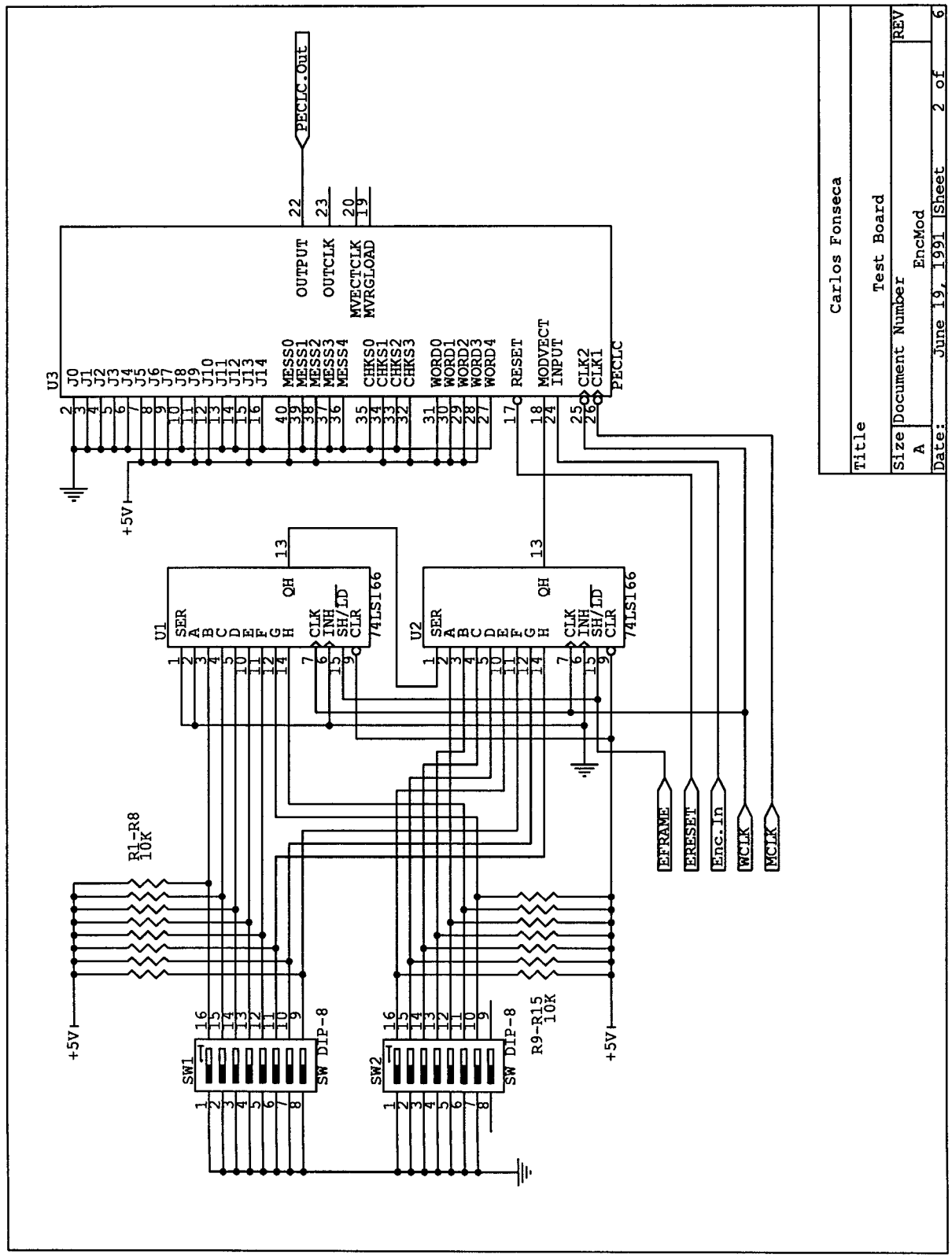


Figure D.2: Encoder module

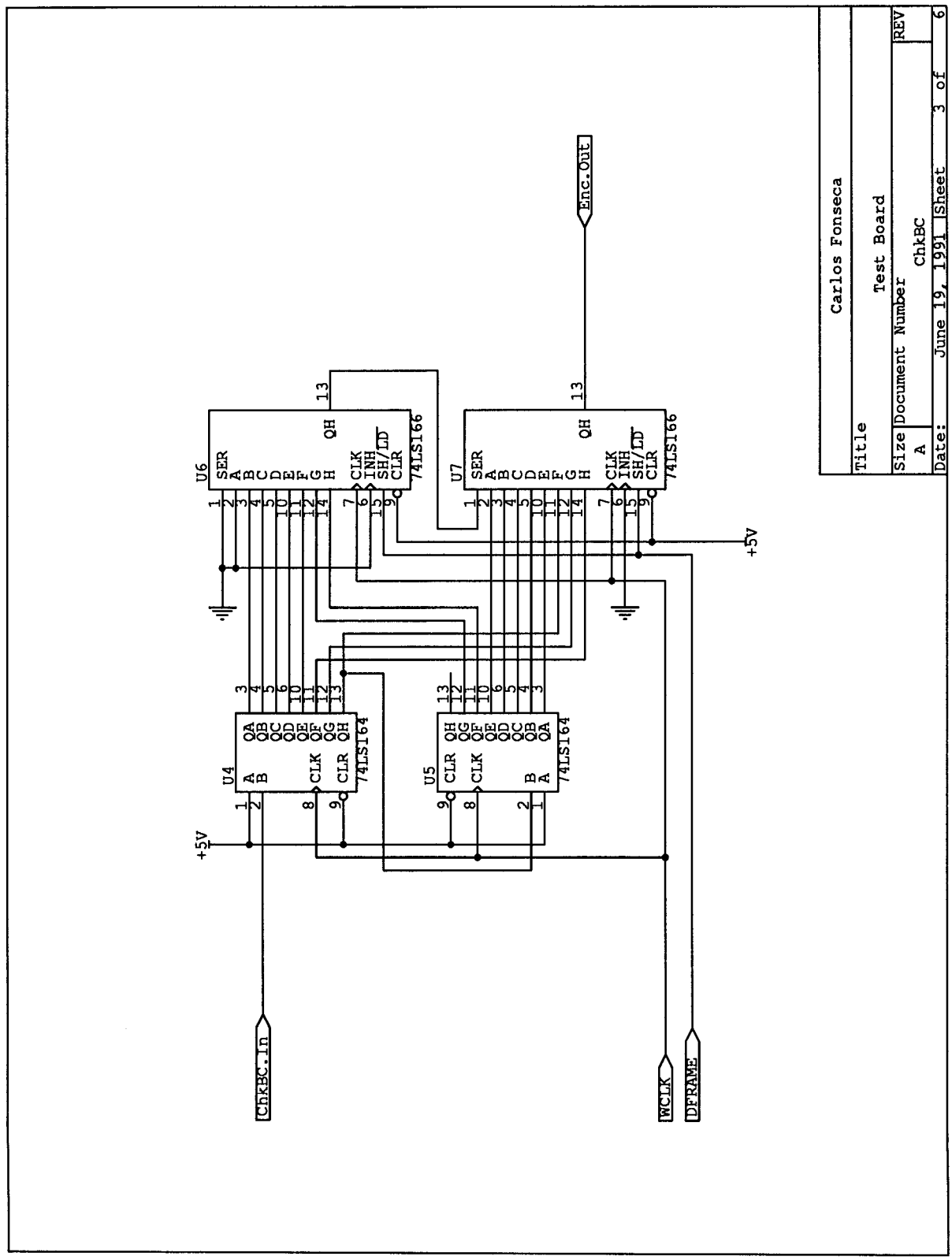


Figure D.3: Check bit order correction circuit

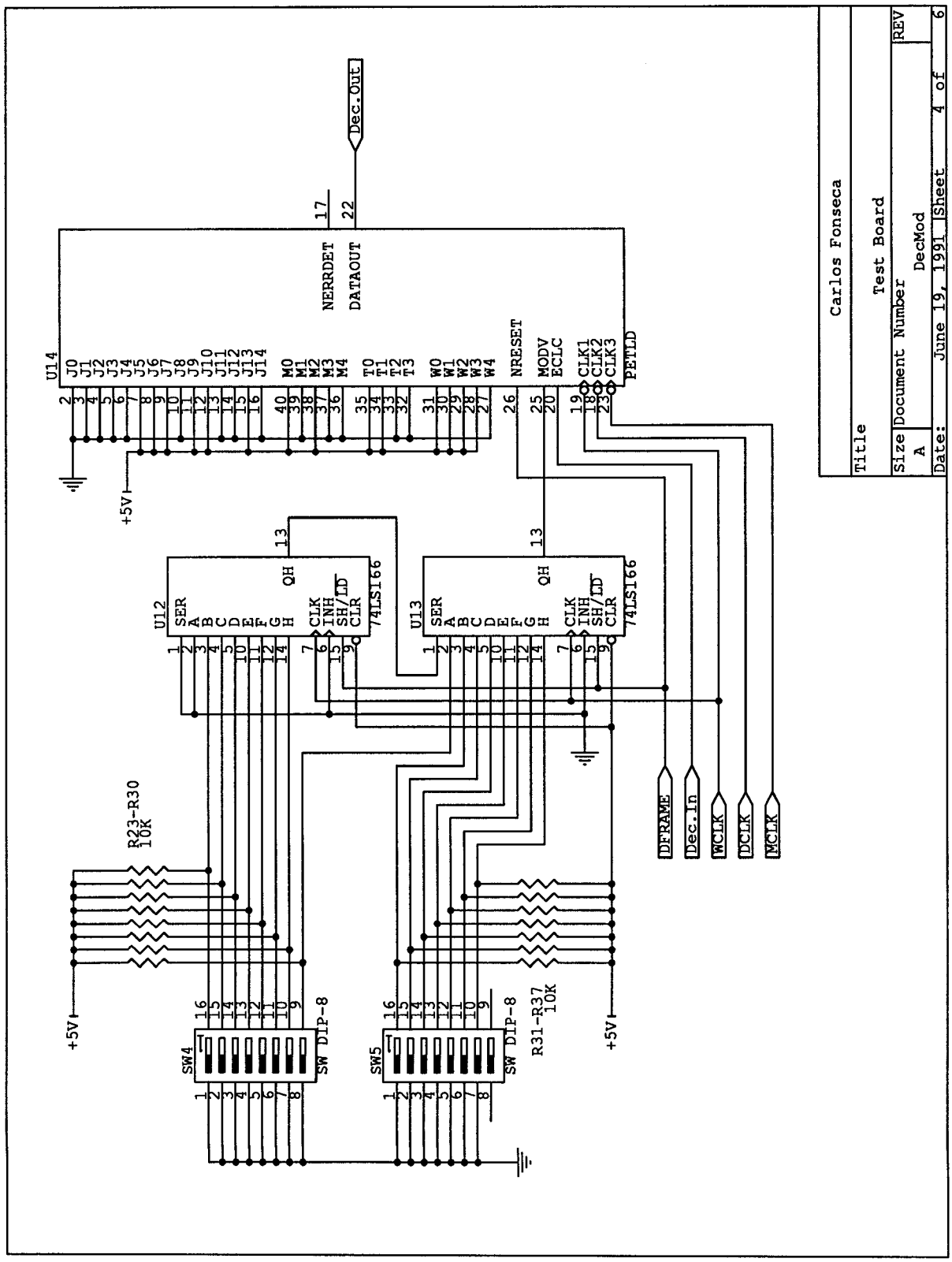


Figure D.4: Decoder module

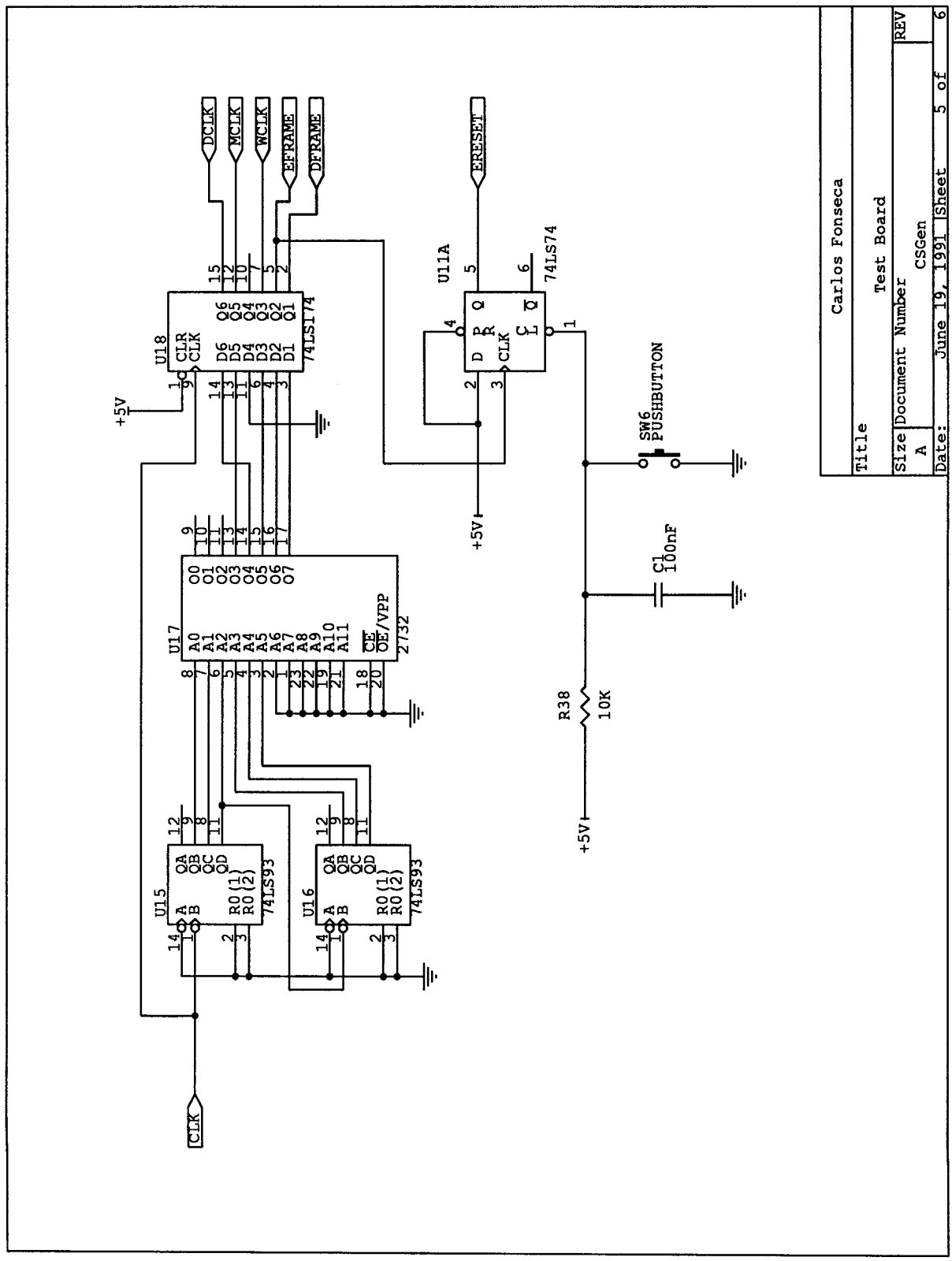


Figure D.5: Control signals generator

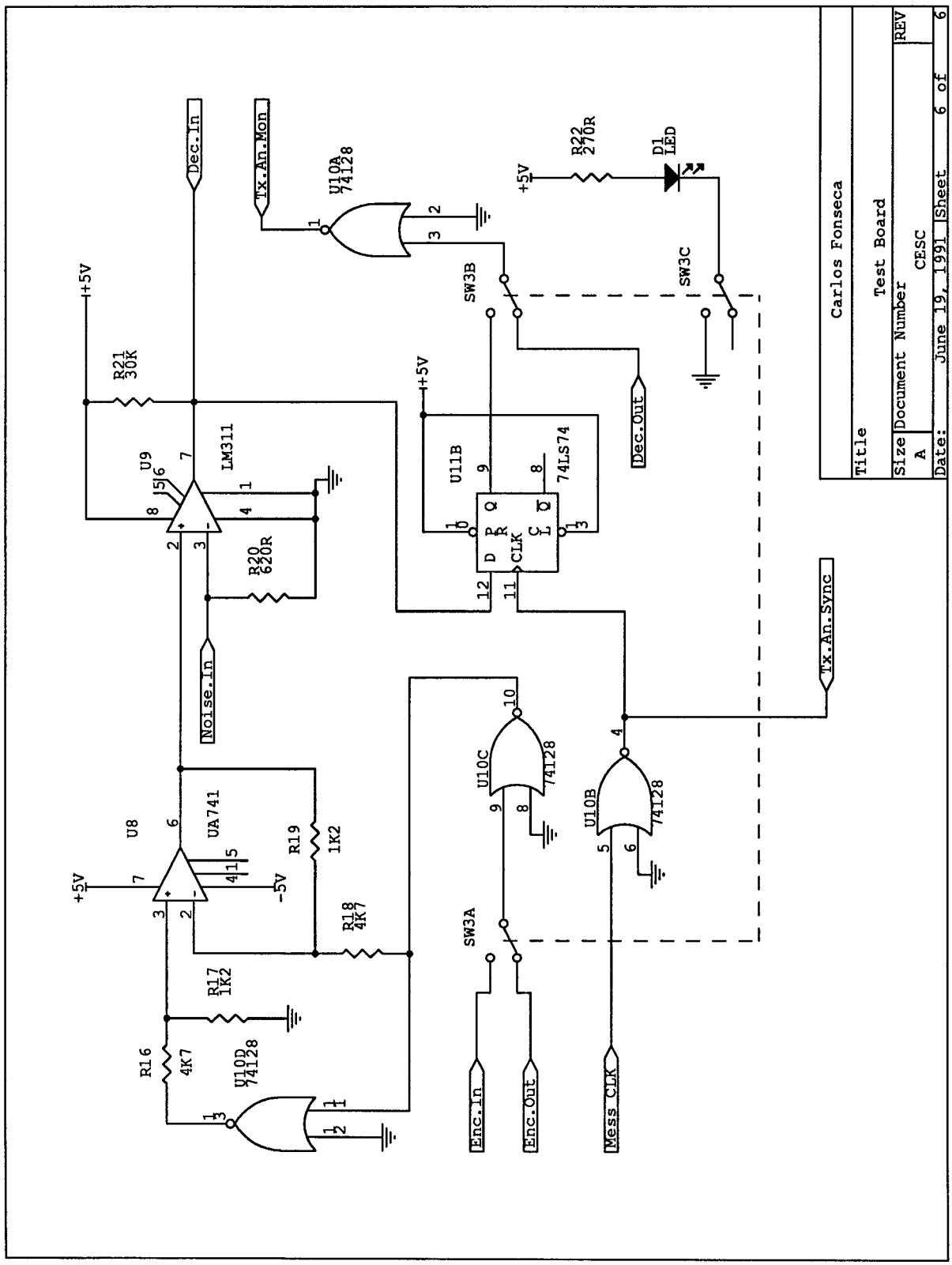


Figure D.6: Channel error simulation circuit

Addr.	Data	Addr.	Data	Addr.	Data	Addr.	Data
00	C7	10	FF	20	E7	30	D7
01	97	11	CF	21	F7	31	E7
02	AF	12	DF	22	C7	32	F7
03	FF	13	E7	23	D7	33	C7
04	CF	14	F7	24	E7	34	D7
05	5F	15	C7	25	F7	35	EF
06	57	16	C7	26	F7	36	EF
07	67	17	D7	27	C7	37	FF
08	F7	18	E7	28	D7	38	CF
09	C7	19	F7	29	EF	39	DF
0A	D7	1A	C7	2A	FF	3A	E7
0B	E7	1B	D7	2B	CF	3B	F7
0C	F7	1C	EF	2C	DF	3C	C7
0D	C7	1D	FF	2D	E7	3D	D7
0E	D7	1E	CF	2E	F7	3E	E7
0F	EF	1F	DF	2F	C7	3F	F7

Table D.1: EPROM contents

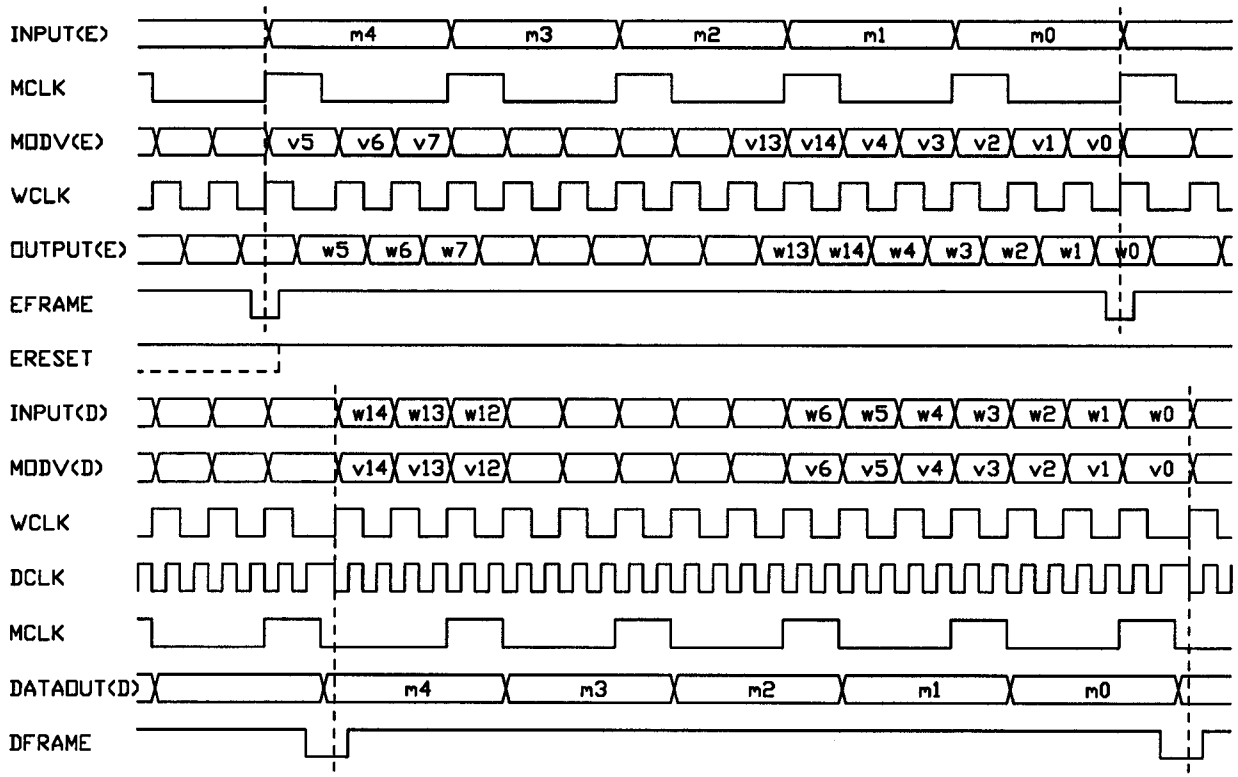


Figure D.7: Circuit timing

Parts list

Test Board Revised: June 19, 1991

Main Module Revision:

Bill Of Materials June 19, 1991 14:54:54 Page 1

Item	Quantity	Reference	Part
1	1	J1	BNC
2	4	SW1,SW2,SW4,SW5	SW DIP-8
3	6	U1,U2,U6,U7,U12,U13	74LS166
4	1	U3	PECLC
5	31	R1-R8,R9-R15,R38,R23-R30, R31-R37	10K
6	2	U15,U16	74LS93

7	1	U17	2732 (EPROM)
8	1	U18	74LS174
9	1	U11	74LS74
10	1	C1	100nF
11	1	SW6	PUSHBUTTON
12	1	D1	LED
13	1	U8	UA741
14	1	U9	LM311
15	1	U10	74128
16	2	R16,R18	4K7
17	2	R17,R19	1K2
18	1	R20	620R
19	1	R21	30K
20	1	R22	270R
21	1	SW3	Ganged Switch
22	1	U14	PETLD
23	2	U4,U5	74LS164