A DIGITAL HARDWARE DESIGN OF AN ELECTRONIC FILING CABINET

by

Michael J. Markowski

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Electrical Engineering in Major

December 1988

A DIGITAL HARDWARE DESIGN
OF AN ELECTRONIC FILING CABINET

by

Michael J. Markowski

Approved: _____

Peter J. Warter, Ph.D.
Professor in charge of thesis

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

This thesis describes the simulations, design, and initial development of an Electronic Filing Cabinet. Originally proposed by Mules several years ago, this begins about where Mules' work stopped.

In Chapter 1, a description is given of the Electronic Filing Cabinet (EFC), a fast document retriever. The chapter is essentially a summary of the system proposed by Mules, though without going into technical details. It is a high level description of how this three-tiered system works.

Chapter 2 details the simulations of two of the three tiers of an EFC. The chapter can be viewed as two major sections: initial simulations to determine how to design the details of the EFC, and simulations of the proposed system. The final system is designed and proposed using the results of the initial simulations. The logical functioning of the system is simulated, not the hardware level. These results are, however, used to design both the hardware and future software system running the EFC.

Like Chapter 2, Chapter 3 is written in two main sections. The first describes the VLSI design done by Shin which makes two of the three EFC tiers. The second half of the chapter describes the system used by the VLSI chips to interface with a PC. Both of these subsystems are discussed at the hardware level without talking about the actual chips used to implement the functions.

Because the two subsystems were designed somewhat independently, Chap-ter 4 details the design of interfacing circuitry which allows for a functioning EFC system

to be built.

Chapter 5 is simply a brief summary of the EFC design with ideas for where future development of the EFC should begin.

CHAPTER 1
THE ELECTRONIC FILING CABINET

1.1 Electronic Filing in the Office

The use of computers has become increasingly widespread in the office. Presently, its chief uses are as interfaces to data bases, storage of local data (i.e., data needed only in a given office, department, branch, etc.), conversion of figures into graphs, word processing, and other work which is inherently computer oriented. However, as the computer further infuses the office place, the routine data generated must be handled in an equally routine manner by the electronic system.

This would of course imply that some sort of inter-office electronic mailing system exists and that the correspondence, when deemed useful, is saved in a data base. How effectively the data is later retrieved depends on the implementation of the data base. There are three basic ways to organize a data base. Two of these are common: "flat" and "inverted".

Probably the most used method is the "flat" organization. Data is grouped into tables identified by the type of entities they represent. Therefore, the existence of a type of document is allowed or disallowed depending on the type and number of these tables in the data base. This limits what documents may be added to the data base and is very undesirable in an office where memos and other correspondence do not always fit neatly into an acceptably small number of categories. The "inverted" data base is quite different in this respect. Rather than relying on predetermined tables, or fields, in a data base, it builds an occurrence list naming every document in which a given word

occurs. While such a system is extremely fast at retrieving documents, the dictionary of occurrence lists usually takes up more space than the document data base itself. As noted by Mules [3], a flat data base system is best suited to finding characteristics of a given document, while inverted systems are best at finding documents with given characteristics.

The third basic way to organize a data base is the least used—almost never. It is a serial search. Documents are retrieved simply by looking through the entire data base until the wanted file or files are found. Serial searching is almost never used on a large scale because it requires dedicated hardware. A general purpose computer is not nearly fast enough to accomplish the task. To quote from Mules [3]:

> [It is estimated that] 3,000 to 4,000 sheets of paper are contained in one file drawer and that each sheet contains 1,000 to 3,000 characters of variable data... A reasonable unit for an office file system would be the equivalent of 10 to 20 drawers, or on the order of 100 million (10^8) characters of coded data.
>
> ⋮
>
> If a 100 million character data base is to be searched in ten seconds, then every 100 nanoseconds the system must absorb a new character from the file source and determine if that character in concert with the previous characters has produced a match with the desired document characteristics.

Certainly, no software system using present or foreseeable technology can possibly meet or even near this speed. But by using special purpose hardware that can work at this high speed, the system can be very flexible and easy to use with the added benefit of having a data base that requires almost no maintenance. Because of the benefits a serial search has over present technology for application to the office filing problem, this is the method, initially proposed by Mules [3], that we have pursued.

1.2 The Electronic Filing Cabinet

Most of the specifications for the Electronic Filing Cabinet (EFC) are taken from Mules [3]. After verifying some of the EFC functioning, we have continued roughly where Mules left off. The system he proposed is designed with a simple three-tiered hierarchy. From lowest tier to highest, the EFC is made up of character matches whose outputs feed into the next tier of string matchers whose outputs in turn feed into the single top level document matcher whose output is the EFC system output.

Central to the design of the Electronic Filing Cabinet is the string matcher. There are probably eight to sixteen such matchers in the full EFC system making up the second tier (an EFC system can be easily expanded since it turns out that one chip will be occupied by a single string matcher). Each of the string matchers can be loaded, or programmed, with a specific string to be searched for. As the character input stream to the EFC passes through the system, the output of a string matcher will either be **true**, signifying a match has been made with some portion of the input stream, or **false**, signifying no match has occurred. The string matcher outputs are used as the inputs to the third and highest tier, the document matcher. When all strings searched for have been located, the document matcher's output becomes **true** signifying that a matching document has indeed been found.

The inputs to the string matcher itself come from the lowest tier, tier one. Every string matcher has sixteen inputs from character matchers. Above, it was mentioned that a string matcher is loaded with a target string. In reality, each character of the target string is loaded into a corresponding character matcher whose outputs go to the string matcher. This is illustrated in Figure 1.1. Then, each character from the incoming data stream is compared to what is loaded in each of the tier one character matchers. Similar to the tier two string matchers, the output of a character matcher is **true** if the input character matches the loaded value, and **false** if it doesn't. The character matchers also have another useful feature. In addition to matching for single

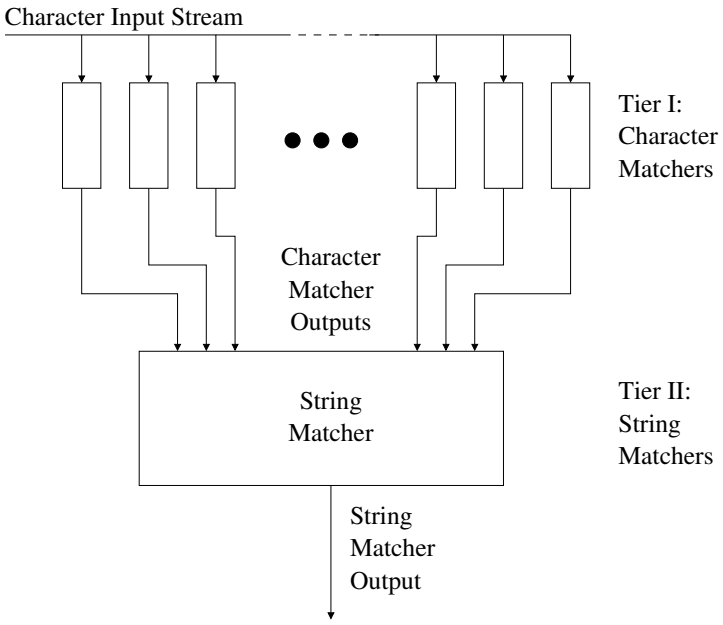

Figure 1.1: EFC Tiers I and II

characters, they can also match for groups. (E.g., all upper case letters, punctuation symbols, "white space" characters, etc.) This gives extra power and flexibility to the EFC system. But because all character matchers are continuously matching against incoming data, the string matcher only considers certain character matcher outputs important at any given time.

All three tiers of the EFC system are shown in Figure 1.2. The interconnections drawn between the string matchers are connective logic. Such an organization makes it possible to preserve the order of occurrence of the target strings. For instance, the second string matcher will begin actively searching only after the first string matcher has located its target. This connective logic is also implemented with counters so that one word may be separated by a given number or range of words. Alternatively, the logic need not be used at all so that all string matchers are always actively searching. Searching in this manner makes the order of occurrence of the target strings unimportant.

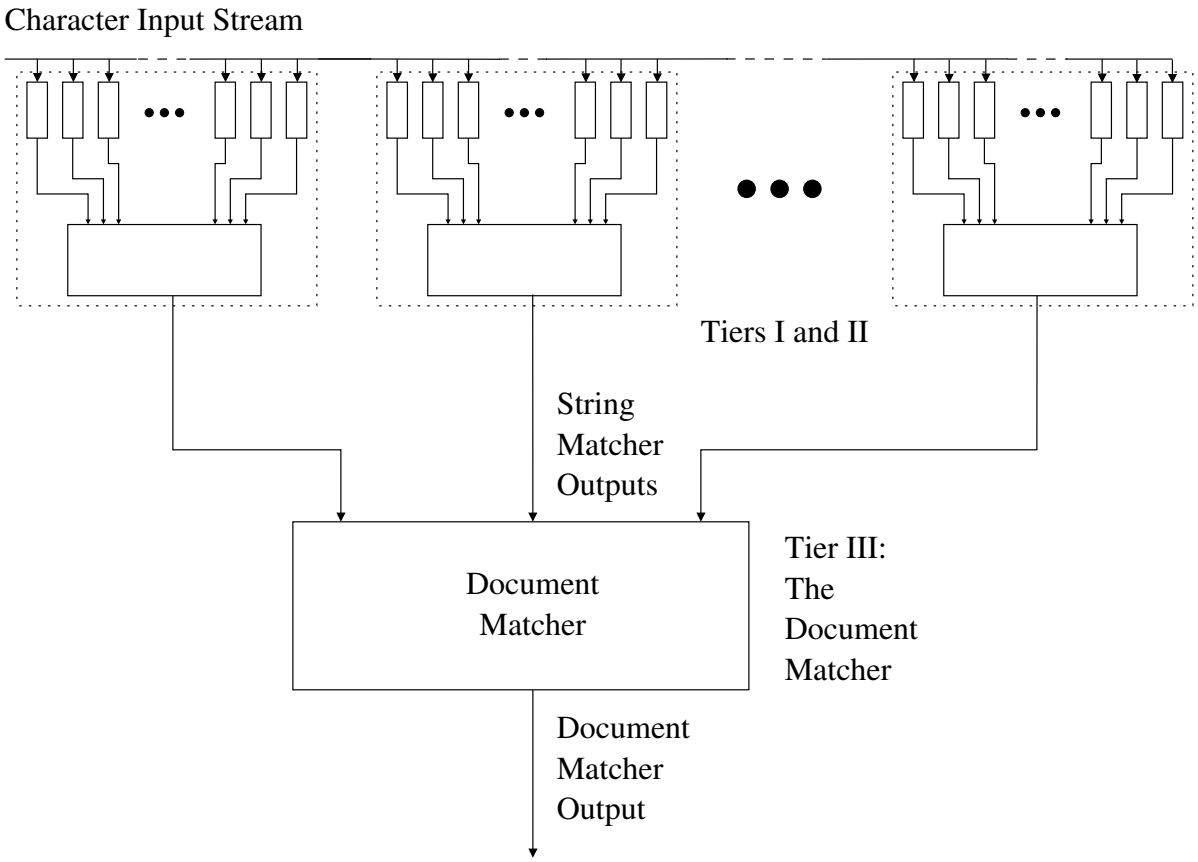

Figure 1.2: EFC Tiers I, II and III

A	B	C	D	E	F	G		Target
A	B	D	E	F	G			Deletion Error
A	B	**D**	**C**	E	F	G		Transposition Error
A	B	**X**	C	D	E	F	G	Insertion Error
A	B	**X**	D	E	F	G		Substitution Error

Table 1.1: Typographical Error Types

The connective logic is implemented by introducing four more character matchers. Two of the matchers are in Don't Care cells and two are in Chip Select cells. In the Don't Care, or DC, portion, there is one character matcher called the Active, or ACT, cell and one called the Delay, or DLY, cell. Each has an associated counter. After a string matcher has made a match, the DC-DLY is activated. As the data stream comes in, the DC-DLY will match it against the character stored there. Each time a matching character goes by, the DC-DLY counter will be decremented. When it reaches zero, that will activate the DC-ACT cell which is identical in functioning to the DC-DLY. When the DC-ACT counter reaches zero, though, it sends out a signal which will enable the next string matcher to begin searching.

The Conditional Search (CS) Cells work in a similar manner but at a higher level. Also, there are no counters in the CS cells. One character matcher is the Enable and the other is the Disable matcher. The entire string matcher is disabled until a character arrives in the data stream which matches what is loaded in CS Enable. Similarly, the string matcher is disabled whenever a character matches what is in the CS Disable cell. This connective logic will be discussed more in Chapter 3.

One of the more useful functions of the EFC is its error tolerance. In normal office correspondence typographical errors will be relatively common, and it must be possible to locate documents containing errors. The EFC can identify four types of errors which are shown in Table 1.1 from Mules [3]. The example of a deletion error

shows a string identical to the target except for the missing “C”, and the trans-position error example has the “C” and ”D” transposed. Conversely, the insertion example shows an “X” added to the string, and the substitution error has an “X” incorrectly in place of the “C”.

As input arrives in the EFC, string matchers are expecting characters to arrive in a certain order so that a match might be made. When a perfect match occurs, that is straightforward enough for the EFC to manage. However, when an unexpected character arrives, the EFC must make a “note”, by setting an error flag, that an error has occurred. In addition to determining that there is an error, the EFC must also find out which type of error occurred. By observing the sequence of events when an error occurs, an algorithm was developed (and used by both Mules and us) which will match all five strings in Table 1.1 with the target.

To implement this error tolerance in hardware, circuitry is required which is dedicated to advancing through a state machine which detects and tolerates certain errors. Because the string matcher allows for errors as it is continually searching for a matching string or token, it can never know when a matching token has begun. Therefore, several identical token searching circuits, which we call token followers, must be available to find a matching token. A token follower will be activated on every incoming character and assume that each character is the first one of the target token. In most cases, this will not be **true** and the token follower will be quickly deactivated. Enough token followers must exist so that one will usually be available when a character arrives in the EFC. This problem, along with the algorithm itself, will be discussed more fully in subsequent chapters.

1.3 The Design of an Electronic Filing Cabinet

Much of the pattern matching and error detection can be done in parallel in the EFC system being proposed. Therefore, VLSI is the obvious choice for reasons of

speed and cost. The design and development of the VLSI implementation of the EFC was undertaken by Shin [4]. Shin describes in detail the physical level functioning of the Electronic Filing Cabinet.

Beyond choosing the technology with which to build the EFC system, there are many decisions to make dealing with the functioning of subsystems within the EFC. As with most systems, the goal is to obtain the best performance from the cheapest and simplest design. This means deciding upon control mechanisms, amount of hardware required, complexity of design—both logical and physical, area required for VLSI design, etc. Some of the specific issues to be addressed are:

- Usefulness of Error Tolerance Algorithm. While in theory having error tolerance is desirable, it must be determined if it is practical. If the tolerance is too much then not only will simple misspellings be matched, but even unrelated words will be seen as matching each other. (E.g., "desk" and "mask".) Conversely, a certain amount of mismatching is unavoidable since, for instance, the word "ate" is contained in the word "heater". To determine the error tolerance usefulness, the following information is also needed:

  1. Mismatch Occurrence. Find out what is necessary to strike a good balance between matching misspellings but not having too many mismatches.
  2. Errors Encountered by Token Followers. Find out how many errors were encountered in the matching string. If too few words match with zero errors, then the tolerance must be made stricter, but if extremely few errors are encountered, the tolerance will be increased.

- Minimum Number of Token Followers Required. Ideally, every word in the input that matches the target should be seen as such by the EFC. It is not clear, however, how many token followers are necessary so that this is the case. Because of the complexity of a token follower, finding the minimum required is desirable.

- String Matcher Performance. The string matcher is made up of several token followers and character matchers. Once the number of token followers has been determined, the work load distribution among the token followers must be known. Not only will this help in performance evaluation, but it will show how likely it is that a match will not be found when it should. The only reason this would happen is if the tolerance to error is so great that all token followers are active and none are available to notice that a target word has arrived.

There is no easy way to solve these problems using paper and pencil. The only reasonable approach is to simulate the system functioning under various conditions. While doing this, statistics describing the important areas of the EFC performance should be gathered. This is precisely what we have done. Subsequently, I will describe what was simulated and what statistics were gotten. From these results, the final determination of the subsystem design will be made and simulation results given under those new conditions.

CHAPTER 2
THE EFC SIMULATIONS

2.1 Design of the Simulator

The simulator of the Electronic Filing Cabinet is used to obtain results which will make clear how to design the system with sufficient hardware and software so that it will work well. Because it is already clear how to design certain aspects of the EFC system, there is no need to simulate the entire system of string matchers and associated connective logic. It is, however, necessary to use simulations to determine how complex the logic associated with a single string matcher must be. The supporting logic can be neither too small and thus overburdened nor too large and little used. Therefore, a single string matcher has been simulated. This initial string matcher is designed with ten associated token followers—clearly many more than would be needed. Also, regardless of what the word length is, a maximum of three errors are allowed per matching word. This, again, is obviously more than one should allow in, say, a three or four letter word. However, by looking at the results of the simulations of a system more tolerant than desired, it will make clear how to design the EFC system as simply as possible with the best possible performance. The following descriptions of the EFC system are based on Mules' *An Electronic Filing System* [3] which contains more detail on the portions of the system which were not simulated.

2.1.1 The EFC Subsystem Simulated

In an EFC string matcher there are two major pieces of hardware: the character matcher and the token follower. A single character matcher is straightforward to simulate. It will have been loaded with a single character or character group and will compare an incoming character to it. If it is the same, the character matcher's output is true otherwise the output is false.

The heart of the string matcher, the token follower, is more difficult. The token follower is, in fact, the most logically complex component of an EFC chip. A new token follower, if available, is activated on every incoming character. As each character arrives, the token follower will advance one state through a state machine which allows for simple typographic errors during the pattern matching. The truth table for this algorithm is shown in Table 2.1. The state machine described has three inputs, four outputs, and two internal variables. The inputs to the state machine are outputs from the character matchers corresponding to the expected character, the next expected character, and the second previous expected character. A token pointer points to the expected character in the string matcher. The other character matcher outputs will be ignored until the token pointer is updated to point to a new character matcher. This is illustrated in Figure 2.1. When a token follower is activated, this token pointer always points to the first character matcher. In the hardware, all character matchers receive the incoming character as input, but the token follower only looks at the three outputs needed by the state machine.

The two internal variables, ef and ref, are error flags. Ef is set whenever a deletion or transposition error is detected, and ref is set whenever an insertion or substitution error occurs. Initially both flags are not set and remain unset as long an error-free match occurs. But if an incoming character matches the next, not the present, character of the target string, ef is set meaning that a possible transposition/deletion occurred. Similarly, if an incoming character does not match any expected character or

Inputs					Outputs					
External			*Internal*		*Internal*		*External*			
i	i+1	i-2	ref	ef	ref	ef	i1	i2	ie	ab
0	0	0	0	0	1	0	0	0	0	0
			0	1	1	0	0	0	0	0
			1	0	0	0	0	0	0	1
			1	1	0	0	0	0	0	1
0	0	1	0	0	1	0	0	0	0	0
			0	1	0	0	0	0	0	0
			1	0	0	0	0	0	0	1
			1	1	0	0	0	0	0	1
0	1	0	0	0	0	1	0	1	1	0
			0	1	0	0	0	1	1	0
			1	0	0	0	0	1	1	0
			1	1	0	0	0	0	0	1
0	1	1	0	0	0	1	0	1	1	0
			0	1	0	0	0	0	0	0
			1	0	0	0	0	1	1	0
			1	1	0	0	0	0	0	1
1	x	x	0	0	0	0	1	0	0	0
			0	1	0	0	1	0	0	0
			1	0	0	0	1	0	1	0
			1	1	0	0	0	0	0	1

i - expected character
i+1 - next expected character
i-2 - second previous expected character

ef - Error Flag
ref - otheR Error Flag

i1 - increment token pointer by 1
i2 - increment token pointer by 2
ie - increment error counter by 1
ab - abort search

Table 2.1: Error Tolerance Algorithm Truth Table

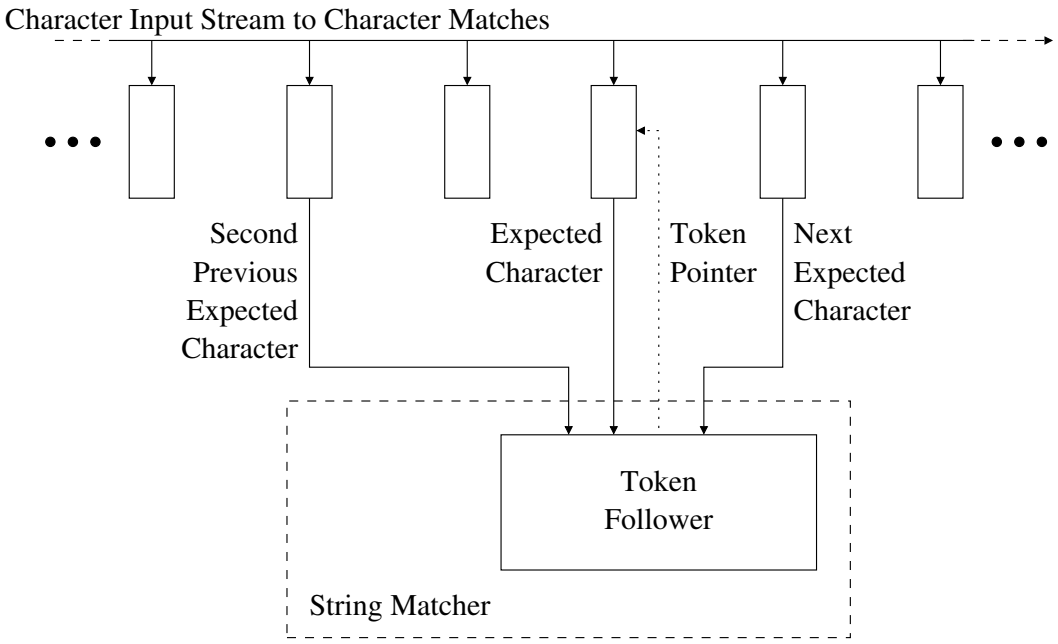

Figure 2.1: The Inputs of an EFC Token Follower

matches the second previously expected character, then a substitution/insertion error is assumed, and ref is set.

The four output variables are to used increment token pointer by one, increment token pointer by two, increment error counter, and abort search and deactivate token follower. Incorporated into the error tolerance truth table, a search is aborted whenever two consecutive errors are encountered. The error counter keeps track of how many non-consecutive errors have occurred. If this count reaches a preassigned limit, the search will terminate.

2.1.2 The EFC Software Model

The EFC string matcher simulator was written in the C language using modern design methodology. All subroutines are fairly small and perform only one simple function rather than using large, overly complex routines. By designing in this manner,

```
struct ch_matcher {
        char *ch_lst; /* list of characters that can be matched */
};
```

Figure 2.2: Character Matcher Data Structure

```
struct tok_follow {
        struct state *current_st;   /* current state number */
        short err_cnt;              /* total number of ref and ef occurences */
        short ss_cell;              /* cell that token pointer is pointing to */
        short ef;                   /* 1 ⇒ deletion/transposition has occurred */
        short ref;                  /* 1 ⇒ insertion/substitution has occurred */
        short abort;                /* 1 ⇒ short search */
        short active;               /* 1 ⇒ token follower is active */
```

Figure 2.3: Token Follower Data Structure

the type of inputs and outputs of a subroutine can remain the same while allowing the routine itself to be modified whenever necessary without requiring any redesign of other parts of the program.

The simulator acts on its input by activating token followers and moving them through the error tolerance state machine. Clearly then, the speed of the simulation depends in large part on how quickly token followers can be advanced to new states. Therefore, we decided that the simulator should be table driven rather than using a conditional branching tree. At startup, a representation of the state machine is generated using linked lists. Once created, when token followers are advanced to a new state, it is necessary only to update a pointer into the state machine data structure.

As mentioned previously, simulation of the character matcher is very simple. The data structure for the character matcher is shown in Figure 2.2. The only field in the structure is a pointer to a list of characters which usually will contain only a single character. The reason for allowing more than one to be stored is that in hardware

```
struct str_matcher {
        struct ch_matcher reg_matcher[16];
        struct tok_follow tok[10];
        int str_len;
};
```

Figure 2.4: String Matcher Data Structure

it will be possible to have don't care bits such that the difference between upper and lower case characters is ignored, etc. The easiest way to simulate this in software is just to put, in this example, the upper and lower case characters in the list.

The data structure for the token follower, shown in Figure 2.3, is more involved than that of the character matcher. The seven fields correspond to the functions carried out by the token follower described, except for the *current_st* field, in detail in Section 2.1. This is a straightforward mapping of the desired capabilities into a data structure. The *current_st* field is a pointer into the representation of the error tolerance state machine. Just as the token follower logically operates in hardware, the *current_st* pointer is advanced through the state machine depending on its input from the character matchers and the two error flags.

With the two previous data structures created, the string matcher data structure can now be designed. All that is necessary is to pull sixteen character matchers into an array to create the full string size, and to add the token followers. The data structure is shown in Figure 2.4. The *str_len* field is the string length of the target string loaded into the character matchers. When the token pointer, or *ss_cell* of the token follower data structure, is greater than *str_len*, a match has occurred.

2.1.3 EFC Statistics Gathered

After the EFC simulations affirm the correct functioning of the system, the most important information the simulator provides is the various statistics about what goes

on internally to the EFC. For the statistics to be correct, however, the simulator input must be an appropriate representation of what the EFC will receive under normal conditions. We use the Kucera-Francis word list of the 12,224 most commonly used English words of at least four characters. These words were obtained from a random sample of 1,000,000 words. Associated with each word on the list is an integer representing its relative frequency in the sample. By using this word list along with the frequencies of occurrence, statistics can be gathered in several important areas:

- Mismatches—when a correctly spelled word is mistakenly identified as a misspelling of the target word.
- Token Follower Usage—how many token followers are needed to make a match with the target.
- Typographic Errors Encountered—how many errors are encountered by the token follower which makes the match with the target.

In each of the cases, statistics were obtained both for words of specific lengths and for all words. (E.g., mismatch statistics for all n letter words and mismatch statistics for words of any length.) For the case of token follower performance, it is, of course, necessary to gather results for overall system performances.

In the following subsections are given the equations used in the gathering of the statistics. To make the equations more concise, I have used a function, $\mathrm{foc}(w)$ for a word w, which returns the relative frequency of occurrence of a word.

2.1.3.1 Mismatch Statistics

After a mismatch has occurred, we would like to determine the probability of it happening again. Since the probability of a certain word being the target is independent of that of a word being the input, the probability of the mismatch is

merely the product of the probabilities of each of their occurrences. In the case of these simulations, frequencies of occurrence are provided, not probabilities. So that to determine the probability of a word occurring, it is necessary to divide the frequency by the total number of words in the sample. To find the total probability of mismatches for a set of words, each of the individual mismatch probabilities is summed together. This is shown in Equation 2.1.

$$\text{Prob(mismatch)} = \frac{\text{foc(target) foc(input)}}{[\sum_{\text{all}\,x} \text{foc(word}_x\text{)}]^2} \tag{2.1}$$

where word_x, is the x^{th} word in the Francis-Kucera database and *target* and *input* are some word_m and word_n, in the data base.

2.1.3.2 Token Follower Usage Statistics

It is slightly more involved to find token follower usage statistics. After an arbitrarily long word matches with an n letter target, the frequency of occurrence of the matching word is determined along with the number of token followers it used to make the match. This is the numerator in Equation 2.2. To obtain the denominator, the sum of occurrences of all words which may take *any* number of token followers to match an n letter target is computed. This division, shown in Equation 2.2, yields the probability of the word requiring i token followers to match then letter target word.

$$\text{Prob(wd}_{n,i}\text{ occurring)} = \frac{\text{foc(wd}_{n,i}\text{)}}{\sum_{i=1}^{10} \sum_{\text{all wd}_n} \text{foc(wd}_{n,i}\text{)}} \tag{2.2}$$

where $\text{wd}_{n,i}$ is an arbitrarily long word which uses i token followers to match the n letter target string.

2.1.3.3 Typographic Errors Statistics

Finding the statistics dealing with typographic errors encountered is almost identical to that of finding the statistics on token follower usage just discussed. When an n letter target is matched by an arbitrarily long word, the number of typographical errors encountered by the *matching* token follower, not any of the others which may have been active, is noted. Then, as above, the frequency of occurrence of the matching input word is determined and used as the numerator. The denominator is the sum of *all* words which matched n letter targets regardless of how many errors the matching token followers encountered. This is expressed in Equation 2.3.

$$\text{Prob}(\text{wd}_{n,j}) = \frac{\text{foc}(\text{wd}_{n,j})}{\sum_{j=1}^{10} \sum_{\text{all wd}_n} \text{foc}(\text{wd}_{n,j})} \tag{2.3}$$

where $\text{wd}_{n,j}$ is an arbitrarily long word which encounters j errors in the token follower that matches the n letter target string.

Each of the three previous equations gives the probability of the event occur ring for just those target and input strings. When total probabilities are wanted for various sets and subsets of the data base, then each of the individual probabilities are added together.

2.2 Results of the Simulator

While it is fairly straightforward to decide on the basic components necessary to implement the Electronic Filing Cabinet in VLSI, it is not nearly as easy to decide on the number of components nor on the details of their functioning. It is not clear, for instance, how many token followers should be available such that on average there will be neither too few nor too many token followers. Also, the error tolerance scheme has to be simulated to determine its effectiveness. The goal is to have the EFC system match words with simple typographical errors but not match with too many correctly

spelled non-target words (as if they were misspelled targets). The simulations help provide answers to these questions.

The data files used by the simulator are subsets of the Kucera-Francis word list. To get an idea of how the EFC would perform under different conditions, the simulator is run using every 10 words from the Kucera-Francis word list which is approximately 12,000 words long. Three data base subsets are used: every 10 words starting at the first word in the word list, every 10 starting at the fourth word, and every 10 starting at the seventh word. Also, words of three characters or less are ignored as are words whose frequencies are less than 10 occurrences per million sampled.

2.2.1 Mismatches

To determine the effectiveness of the error tolerance algorithm, the three subsets as described above were used in the simulations. In this way we could know approximately how many incorrect matches, or mismatches, occur that ideally should not. The mismatch problem obviously can never be eliminated in our implementation if only because of the spelling rules of Western languages. Small "words" can be embedded in larger words or correctly spelled non-target words can be very similar to the target of the search and will be seen as misspellings.

The results of Figure 2.5 are obtained when up to three non-consecutive errors and no two consecutive errors are permitted when searching for the target string. Perfect matches are not considered (i.e., a word matching against itself) but only mismatches where a non-target word is mistakenly identified as a misspelling of the target. As expected, Figure 2.5 shows that the shorter words are most prone to being mismatched. Of course, it is not expected in the normal course of operation that short words would even be the target of a search. But when the number of errors allowed approaches or exceeds the target word length, mismatches will increase dramatically as shown.

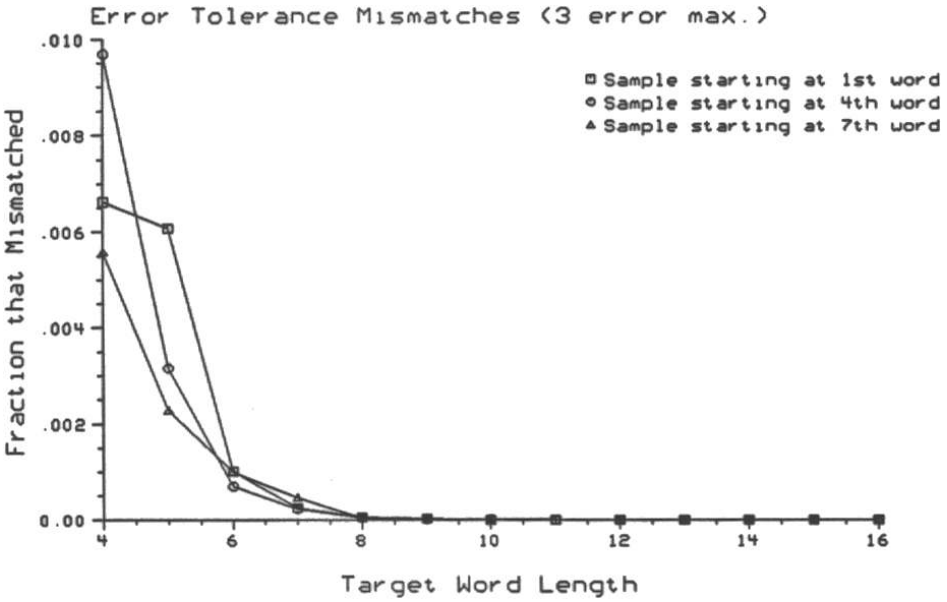

Figure 2.5: Mismatches versus Word Length

By using the loadable error counters provided in the EFC chip, the software running the system can set the counters so that, in effect, the curves in Figure 2.5 are flattened towards zero as much as desired while still allowing misspellings to be matched. As is seen in the figure, when word length increases beyond seven or eight characters there is virtually no mismatch problem. Again, this is what one would expect because the longer a word is, the less chance there is that another similarly spelled word exists. Of course, there are many exceptions to this when words with similar stems are compared with the target.

2.2.2 Token Usage

In determining how many token followers should be part of a string matcher in hardware, the simulator was run using the same three Kucera-Francis subsets as previously described. With the relatively limited space available on the EFC chip, we want to be sure to use an appropriate number of token followers per string matcher.

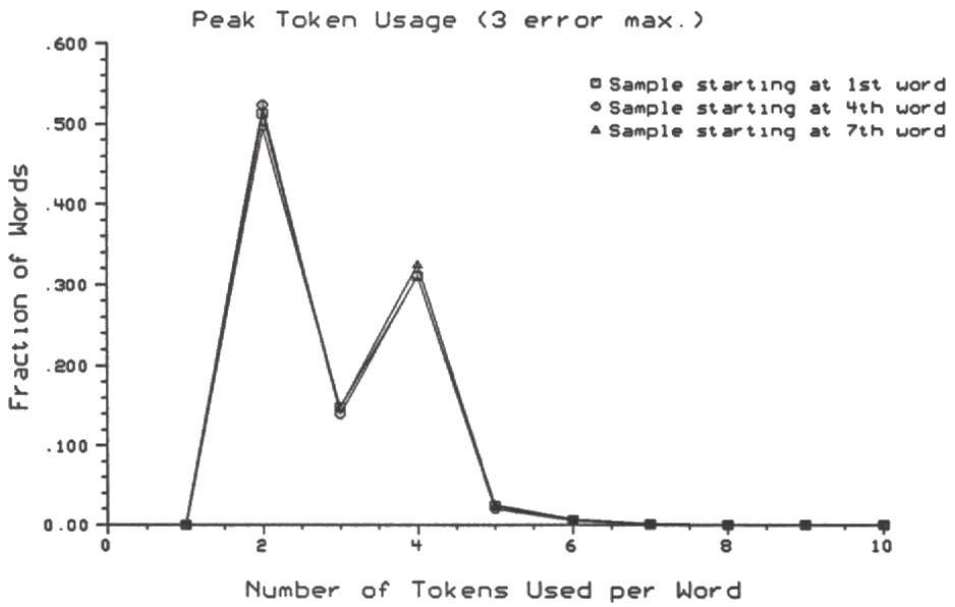

Figure 2.6: Peak Number of Tokens Used in Matching Words

When a perfect match occurs, a minimum of two token followers will be active during most of the search, but only if the first token follower activated happens to be at the first character of the word. Therefore, enough token followers must be available to guarantee that there will almost always be enough to find the perfect match even if other token followers are active when the first character of the target arrives. But that would be the very minimum number of token followers because it does not take into account searches for any non-matches. The results of the simulation can be seen in Figure 2.6.

As before, a maximum of three non-consecutive errors and no two consecutive errors were allowed in the simulation. In this case, though, perfect matches and mismatches were both considered. These results are representative only of matches, then. When a word is being searched for, if the search is aborted then no statistics about token follower usage are gathered for that search. There can be no search which uses only one token follower as is shown in Figure 2.6. This is because if a token follower is

available, then one is activated on every incoming character. At some point during a search for a word, at least two token followers will be active. An easy illustration would be to imagine a token follower activating on the first character of what will become a perfect match. When the second character arrives, a second token follower will be activated and in fact will find a matching token with one deletion error—namely, a missing first character.

The majority of matched words, as shown in Figure 2.6, require the string matcher to use two, three, or four token followers in making the match. However, the figure also shows that although it is a very small fraction, it is possible to require up to nine token followers! This does not mean, though, that the ninth token follower was the one that matched. It means only that at some point during the match, nine token followers were simultaneously active.

2.2.3 Error Tolerance

In order to find out how useful the error tolerance scheme is, the simulator checks all token followers after a match has occurred to see how many errors the token follower encountered while matching the target. The simulator was run in the same manner described previously in Section 2.2.2 to match the target string.

The statistics on the number of errors encountered are further broken down by word length. Using this information, several graphs illustrate the probable system performance when one error is allowed, two are allowed, and the maximum of three are allowed. The EFC software designer can use these graphs so that the error counters are programmed as a function of word length. For instance, if it were desired that 80% or more matches are to be made error-free, then by looking at Figure 2.7, it is clear that for words of less than 5 or 6 characters, zero errors would be tolerated. Further, if 15% would then be permitted to match with a single error (implying that 5% match with 2 errors and 0% with three), from Figure 2.8 we see that words between 5 or 6

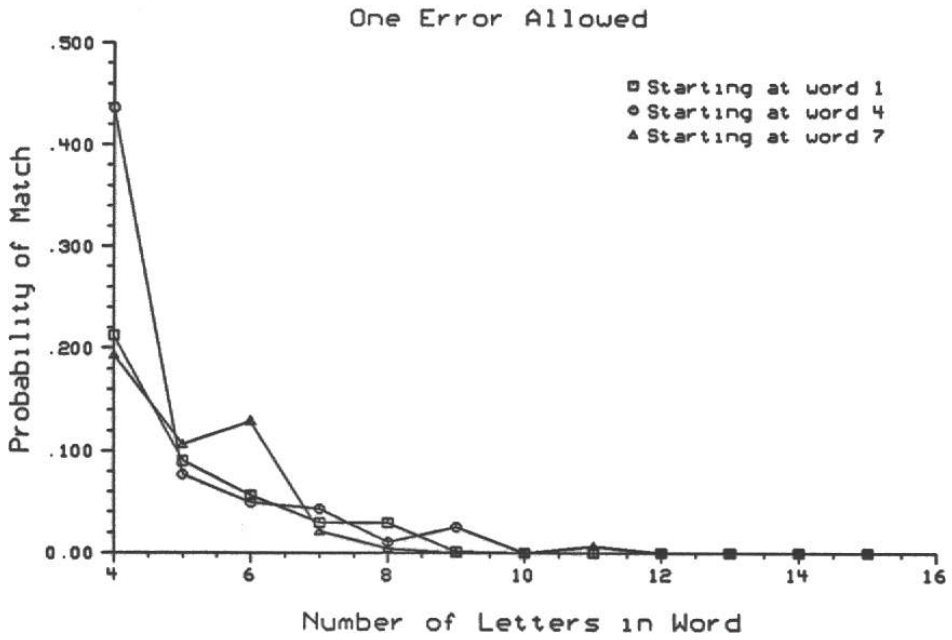

Figure 2.7: Probability of Match with One Error Allowed

and 8 or 9 characters long would be allowed one error. In such a manner, the system software designer can construct a table of errors allowed versus word length which will give the performance he desires.

The predicted system performances from the graphs are, as initially stated, based on simulations using only one string matcher. In normal operation, several string matchers will be used in conjunction with one another to produce more complex data base queries. Although not simulated, one would expect that document mismatches would be much less common than the single string mismatches discussed in this chapter. It is quite likely that good system performance can be achieved even if the error tolerance is rather liberal provided that several string matchers are used in a query.

But overall, if the majority of token followers encounter no errors, then the error tolerance algorithm would be too strict since it would be matching mostly with correctly

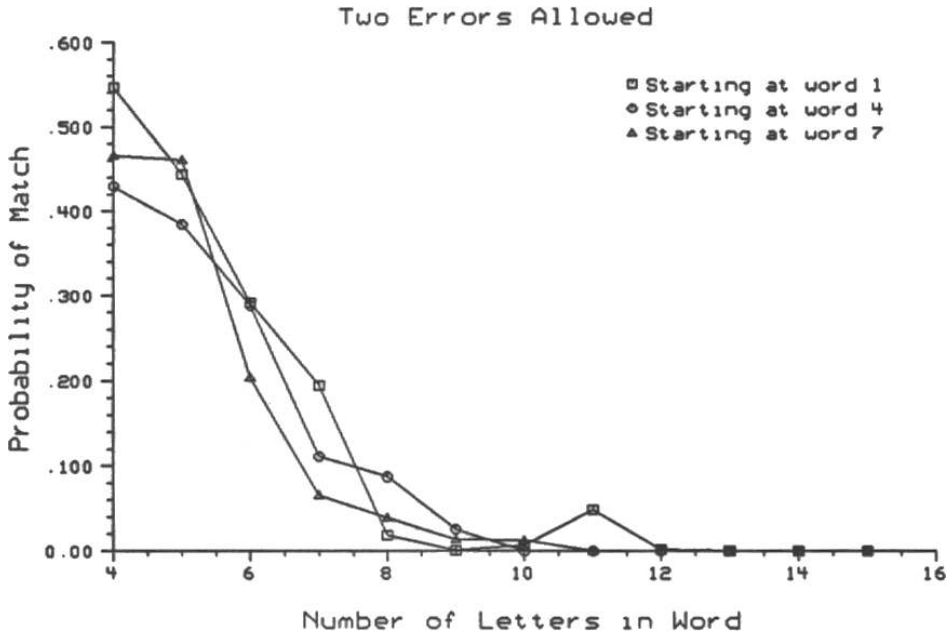

Figure 2.8: Probability of Match with Two Errors Allowed

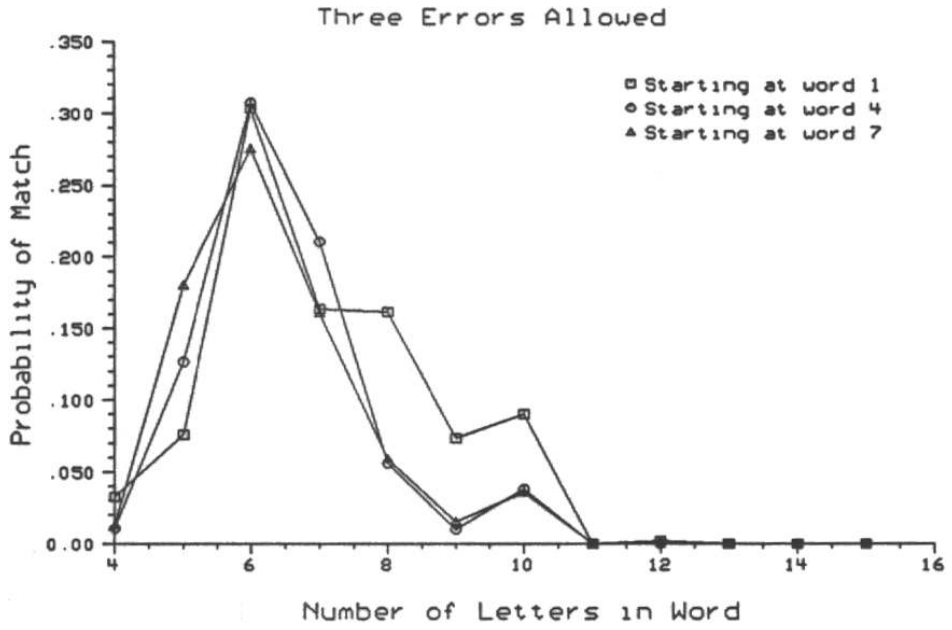

Figure 2.9: Probability of Match with Three Errors Allowed

spelled targets and only with a few misspelled ones. Conversely, if there were many more token followers matching with errors than without, the error tolerance would be too great and, again, not be useful. From Figures 2.7 through 2.8, we see that it is possible to choose a system performance anywhere from one extreme to another. This built-in flexibility should allow the EFC system to be used in a variety of applications.

2.3 Interpretation of Results

One of the more major parts of the simulations was the testing of the error tolerance algorithm. Before any effort is made in deciding how many token followers to use, it must be known with certainty that the algorithm they are implementing is correct. After extensive simulation of the algorithm, only one small defect was found. And this occurs only at the very end of an input data stream. When an error occurs, the algorithm requires that enough data be input so that it can figure out and hopefully recover from the error. Therefore, if the last character of the input is an error or an apparent error, an end state in the state machine will not reached. For instance, if the target string is "house" and the end of the entire input stream, not just the current document, contains "hous", a match which should occur will not. In most cases this will not present a problem, but if a searcher wants to be sure no words are missed he can, through the software, attach a dummy character to the end of the input stream so that any trailing error will be discovered.

Convinced of the correctness of the error tolerance algorithm, we made the simulations described in the previous section. From those results, it is clear that the system does not run optimally under those conditions and that software is required to correct the deficiencies. To summarize the results, less than half of the allotted ten token followers were used significantly, there were too many mismatches against frequently used words, and more errors in general were being tolerated than is desired. Using Figure 2.6, *Peak Number of Tokens Used in Matching Words*, four token followers

Characters per Word	*No. of Errors Tolerated*
1–6	0
7–8	1
9–10	2
11 or more	3

Table 2.2: Number of Errors Tolerated for Different Word Lengths

would appear to be adequate. And from Figures 2.7 through 2.9, Table 2.2 giving the number of errors allowed for words of different length were derived.

When the simulator was modified to work under these new constraints, the same simulations described in the previous section were made using identical input files. These results are detailed in following subsections.

It is important to note, however, that the simulations are somewhat dominated by the more common words. The data base is sorted in order of decreasing frequency. So by looking at the data base after every fifty words and computing the fraction of the data base it represents at that point, the graph of Figure 2.10 is obtained. It shows that the first 55% of the data base contains 90% of the words and that the first 20% contains 70% of the words. So the further the word is in the data base, the less impact it will have on the simulation results.

2.3.1 Mismatches in Modified EFC

The results for target mismatches as shown in Figure 2.10 are clearly too high for short words. But after making the modifications described, the mismatch statistics change considerably and are shown on the graph of Figure 2.11. Note that the maximum Fraction that Mismatched (y-scale) has dropped by a factor of forty. As hoped for, the original curves are considerably flattened near the origin. So that rather than

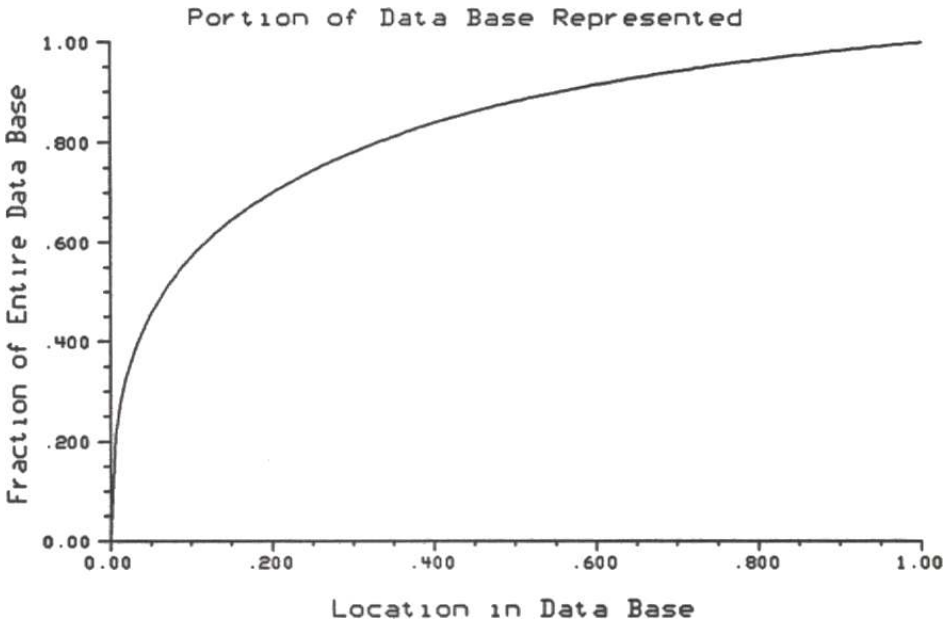

Figure 2.10: Frequency of Sampled Words

the original worst case of one or two four letter words per hundred mismatching, now in the worst case only about two words per ten thousand do. Notice, though, that the curves are flatter but not completely flat. This shows that while fewer mismatches occur, some still do. By nature of the algorithm, this must occur if it is to have any tolerance of errors. The only way the curves could be flat, which would be ideal for mismatches, is to allow zero errors per token, which is definitely not ideal for error tolerance. The goal is to strike a balance between the two, and the statistics for the modified EFC reflect a much better balance than before.

2.3.2 Tokens Used in Modified EFC

One good sign showing that the modifications did not adversely affect the EFC system comes from the data in Figure 2.12. It is similar in some ways to Figure 2.6, *Number of Tokens Used in Matching Words* in that three token followers are used far

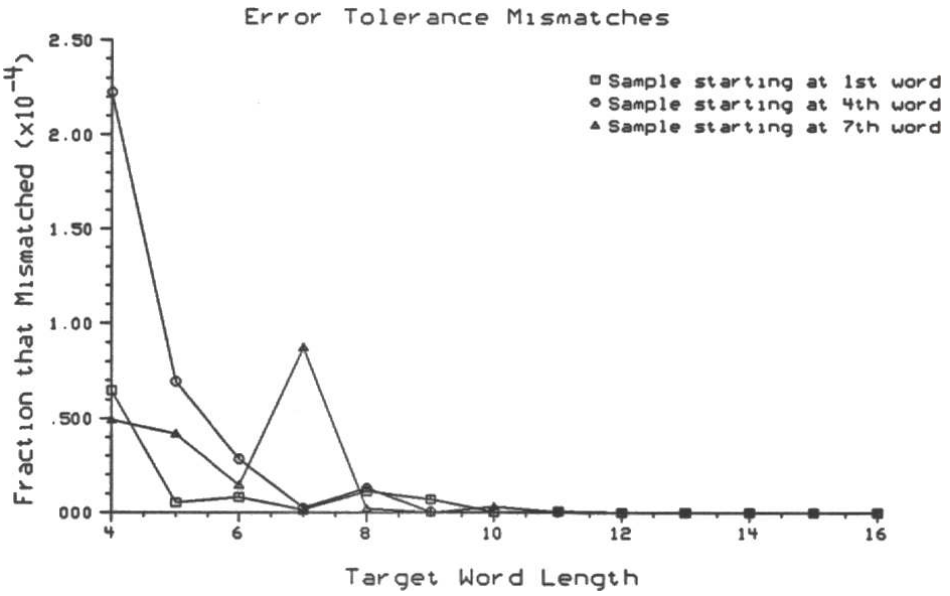

Figure 2.11: Mismatches for Modified EFC

less than two. There is a major difference, however, in that now four token followers are used less of the time than three are. One would hope that as additional token followers are added to the system, the newest would be used less than the previous ones since there usually will be less work for the later token followers. As opposed to Figure 2.6, these new results are very good.

2.3.3 Error Tolerance in Modified EFC

Another problem the initial simulations exposed was that an inordinate number of words with errors were matching targets meaning that too many words were successfully passing through the tolerance algorithm. By modifying how many errors are acceptable, shown in Table 2.2, we got the results of Figure 2.13.

These results are almost ideal. Virtually all matches were perfect matches. It cannot be discerned on this scale, but the last three data points of Figure 2.13 are small

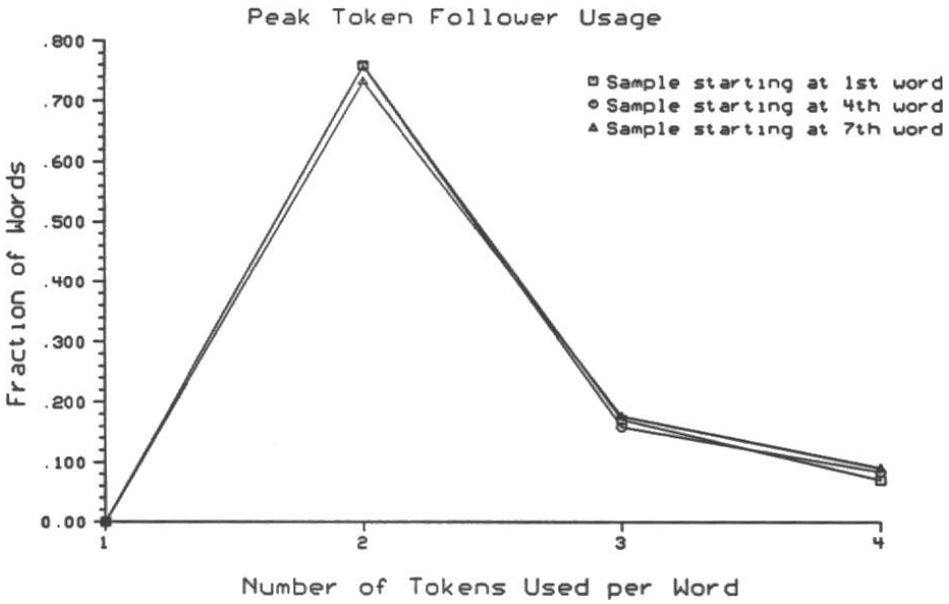

Figure 2.12: Peak Token Follower Usage in Modified EFC

but still non-zero (recall Figure 2.11). While at first it may seem that the tolerance is almost too strict, remember that in the data base there should only be a perfect match when a word matches against itself. That is, there are no duplicate words in the word list.

The simple modifications described in the previous sections show in simulations that they do make a major difference in the simplification of the design of the EFC system without sacrificing performance. Once made, the decision to have four token followers cannot be changed. However, the number of errors allowed per word is completely under software control so that if some uncommon or unthought of use requires it, the table we propose can be ignored entirely and modified as seen fit. This makes for a very flexible system which can tailored to some extent to meet the needs of many applications.

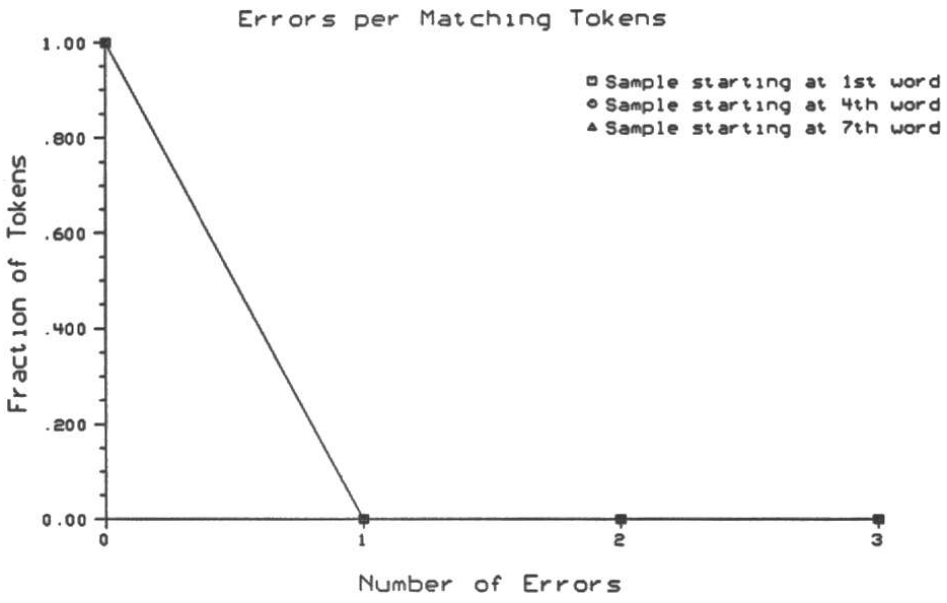

Figure 2.13: Errors Encountered in Modified EFC

CHAPTER 3
THE EFC SYSTEM HARDWARE DESIGN

3.1 The EFC Chip Array

As the design and layout of the EFC functions in VLSI progressed, it became apparent that because of the complexity of the system, an entire EFC system could not be implemented on one chip. The size constraint limited one string matcher with its associated four token followers to a medium size CMOS chip using 3 micron technology. Fortunately, the same parallel functioning which lends itself to a VLSI implementation, also makes it straightforward to split the system functions into several chips. Each string matcher works with almost complete independence from each of the others. The only exception is the connective logic between adjacent string matchers. This inter-chip communication only requires the addition of 2 more pins per chip. The output pin will either enable or disable the next string matcher, and the input pin will, of course, be tied to the output of the previous string matcher. Typical of many VLSI systems, intensive computing goes on within the chip, but relatively few pins are needed for outside communication.

3.1.1 EFC Chip Inputs and Outpus

To communicate with the rest of the EFC system, an EFC chip will have pins for data and for several control signals. A chip with its pins named is illustrated in Figure 3.1. Before a chip begins searching for its target string, that target must be programmed into it along with other data such as error counters, connective logic

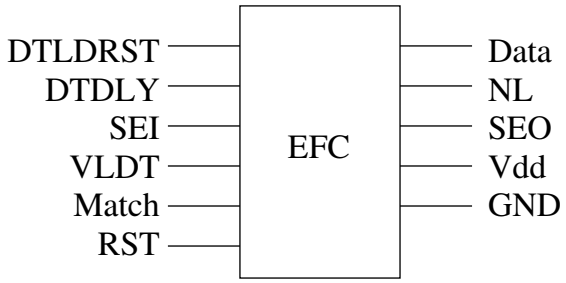

Figure 3.1: EFC Chip Pin Functions

characters and counters, etc. So it follows that some pins on the chip will be used for programming and some for searching. It is worth mentioning that an EFC chip has only two modes of operation program and search. The function of each of the pins is rather straightforward and is defined below:

DTLDRST when asserted, chip is in program mode. When not asserted chip is in search mode.

DTDLY asserted for a short time to prepare all chips in program mode for incoming data

NL attached to DTDLY of next chip.

VLDT alerts the chip that valid data is waiting on data bus.

SEO connective logic output from one chip to enable the next chip.

SEI takes as input the SEO from the previous chip.

Match indicates string matcher has located a matching target in data stream.

RST is used immediately prior to entering search mode to prepare the chip for incoming data.

m	n	Match Condition
0	0	never match
0	1	match a **0**
1	0	match a **1**
1	1	always match

Table 3.1: Character Matcher Match Conditions

There are also pins for 9 bits of data, 2 clocks, and power and ground for a total of 21 pins per EFC chip.

When bytes are sent from the disk to the EFC chips, they are the usual 8-bit bytes. Because it is desired that character matchers be flexible enough to match classes of characters in addition to single characters, the 8-bit characters must be converted into 9-bit characters. With this extra bit, it is easier to separate classes of characters so that only a few bits and don't cares are needed to match against a character class. From this point, to make it easier to distinguish when 8-bit bytes and when 9-bit bytes are being discussed, "byte" will refer to 8 bit data and "byte_9" will refer to 9 bit data.

3.1.2 EFC Programming

Forty-eight bytes_9, are required to fully program a chip. The chip circuitry determines what sequence the data must arrive in, so there is no flexibility in the programming sequence. The first byte_9, is dummy data and is used only for startup. It may be anything since the data bus is not checked at this time. Although the chip has 16 character matchers, it takes 32 bytes_9, to fully represent the 16 characters of the target string. This is because Content Addressable Memory (CAM) is used and represents each bit of the target string using two bits per memory cell. In doing so, it is possible to match against incoming data in any one of four ways. There are two bits, m and n, in a cell. These can be used together to match against incoming data

to always match, never match, match a **1**, or match a **0**. This is shown in Table 3.1.

To load a single character matcher, first a byte_9 representing all its m bits is given the EFC chip, followed by a byte_9 representing all the n bits. This sequence is continued until all 32 bytes_9 have been loaded.

The next 8 bytes_9 are used to control the connective logic. The first 2 bytes_9 represent a character in the same manner as previously described. However, they are loaded in the opposite order. First, the n bits are loaded followed by the m bits. The next 2 bytes_9 are used to load an eight bit counter, but only 4 bits at a time are sent. The four least significant bits of the first byte_9 are used as the four most significant bits of the counter, and the four least significant bits of the following byte_9 are used as the least significant bits of the counter. These four bytes_9 together load what is called the Don't Care Delay, or DC-DLY, cell. Once the string matcher detects a match, in addition to sending a signal outside the chip, it starts this counter counting. The chip continues searching, but now it matches against the character loaded into the DC-DLY cell. Every time it finds one of these characters the counter is decremented. When the counter reaches zero, it activates the next cell, the Don't Care Active, or DC-ACT.

The DC-ACT is loaded with four bytes_9 in exactly the same manner as the DC-DLY and also behaves the same way. The only difference is that when its counter reaches zero, it sends a signal outside the chip, the **SEO**, which will activate the next EFC chip. The DC-DLY and DC-ACT connective logic counters can be loaded with zero so that all chips are searching independently of the others, or it can be used so that after a word is found, searching will not continue until x characters, words, pages, etc. have gone by.

The next byte_9, the error mask, also only uses the four least significant bits. Each bit corresponds to one of the token followers. If the bit is **0**, then that token follower will allow the first letter of a word to be in error (e.g., allow bat for cat). If the bit is **1**, the first letters of both target and data must match. The least significant

bit of the $byte_9$ corresponds to token follower 1, the next to least corresponds to token follower two, and so on. (All simulations described in the previous chapter assumed errors *are* allowed in the first character.) All four of these bits will be **0** or all four will be **1**. Generlly, there should be no reason to have different bit values even though the capability exists.

Following the error mask $byte_9$ is the checker $byte_9$. The four least significant bits are used again but this time for a binary representation of a number. The number ranges from 0 to 15 and represents the number of characters in the target string.

The error counters in all four token followers are set to 0, 1, 2, or 3 using the next $byte_9$. The binary value of the number of errors to allow is stored in the two least significant bits of the $byte_9$. All four token followers, then, are always set to allow the same number of errors.

The third least significant bit of this $byte_9$ is used as the Conditional Search (CS) controller. If this bit is **0**, the connective logic signals SEI and SEO are ignored so that the chip will begin searching according to the chip enable logic described in the next paragraph. Should the bit be **1**, however, the chip still activates according to the chip enable logic but only after it first receives an SEO from the previous EFC chip.

The last four $bytes_9$ of the programming data are used to enable or disable the EFC chip. The $bytes_9$ represent two characters and are loaded in the same manner all characters are loaded. The EFC chip is always disabled until the first of these two characters is spotted in the input stream. When the character is found, the chip is enabled and searching begins. Whenever the second of these two characters is found in the input stream, the chip is disabled and searching halts. However, if both the enable and disable states are simultaneously true, enable has precedence over disable. This precedence can be useful in certain types of conditional searches. If, for instance, a name is to be found in a **To:** field of a memo, the enable character would be the **To:** character code, while the string matcher would be disabled on all field delimiters.

When the **To:** character is received, both enable and disable will trigger. But because enable takes precendence, the chip, or string matcher, is enabled to search.

To summarize using a tabular format, to program an EFC chip the data must arrive in the sequence as shown in Tables 3.2 and 3.3.

When programming has been completed, the EFC chips may begin searching and interacting with the rest of the EFC system. This will be described in Chapter 4.

3.2 The EFC System

While the design of the Electronic Filing Cabinet chip and simulator received much of our attention, the chip is only one part of the system. The entire system is composed of an IBM PC-AT, an Intel 80186 microprocessor, the EFC chips, a Maxtor 175 megabyte hard drive, a disk data controller, and 256K words of RAM through which the different devices may communicate. The memory is read and written by all of the devices except the EFC chips; by nature of their function, they will only read data from the memory. Most of the system underlying the EFC chips was designed by Jacob [2] based on an earlier design by Dr. Peter Warter.

3.2.1 Bus Structure

The communications between the various subsystems of the EFC are sup-ported by two sets of buses. One set is the address and data buses for the 256K words of memory, and the other is the address and data buses for the I/O of the Intel 80186 (or 186 for short). As mentioned previously, the memory is used by all of the major devices in the EFC system. The bus is accessed through output controlled latches which are under the control of various finite state machines. Each device has its own latches leading to the 18 bit wide memory address bus. There are counters in the system, however, which do not have direct access to the bus. The counters are used for

Byte Number	**Description**	**Represents**
1	Dummy Byte match	
2	CAM 0, m bits	Character 1
3	CAM 0, n bits	
4	CAM 1, m bits	Character 2
5	CAM 1, n bits	
6	CAM 2, m bits	Character 3
7	CAM 2, n bits	
8	CAM 3, m bits	Character 4
9	CAM 3, n bits	
10	CAM 4, m bits	Character 5
11	CAM 4, n bits	
12	CAM 5, m bits	Character 6
13	CAM 5, n bits	
14	CAM 6, m bits	Character 7
15	CAM 6, n bits	
16	CAM 7, m bits	Character 8
17	CAM 7, n bits	
18	CAM 8, m bits	Character 9
19	CAM 8, n bits	
20	CAM 9, m bits	Character 10
21	CAM 9, n bits	
22	CAM 10, m bits	Character 11
23	CAM 10, n bits	
24	CAM 11, m bits	Character 12
25	CAM 11, n bits	
26	CAM 12, m bits	Character 13
27	CAM 12, n bits	
28	CAM 13, m bits	Character 14
29	CAM 13, n bits	
30	CAM 14, m bits	Character 15
31	CAM 14, n bits	
32	CAM 15, m bits	Character 16
33	CAM 15, n bits	

Table 3.2: EFC Programming Data Sequence

Byte Number	Description	Represents
34	CAM DC-ACT, n bits	Don't Care Active Counter
35	CAM DC-ACT, m bits	
36	DC-ACT counter, 4 MSB	
37	DC-ACT counter, 4 LSB	
38	CAM DC-DLY, n bits	Don't Care Delay Counter
39	CAM DC-DLY, m bits	
40	DC-DLY counter, 4 MSB	
41	DC-DLY counter, 4 LSB	
42	Error Mask	Stored in 4 LSB
43	Target String Length	Stored in 4 LSB
44	Error Cntr & CS Ctrl	Stored in 3 LSB
45	CAM Enable Chip, m bits	Chip Enable/Disable
46	CAM Enable Chip, n bits	
47	CAM Disable Chip, m bits	
48	CAM Disable Chip, n bits	

Table 3.3: EFC Programming Data Sequence (cont.)

counting addresses and data transfers associated with DMAs and access the bus using registered outputs with output controls. Once on the bus, an address is multiplexed into two 9 bit addresses where these new, smaller addresses are used as the row and column address of the data.

Used in conjunction with the memory address bus is the memory data bus which is 16 bits wide. Each device accesses the data bus through output controlled latches controlled by the same state machines mentioned above.

The other bus set for microprocessor I/O operates a bit differently than do the buses for the memory. Only 8 bits of the address bus are used for I/O, and addresses are decoded into control signals which are routed to the appropriate I/O device. Of these 8 bits, six are sent directly to the disk data controller (DDC) to address one of the 64 on-chip DDC registers. The I/O data bus is 16 bits wide, but unlike the memory data bus, all of the devices do not make use of all data bus bits. All devices on the I/O

bus use output controlled latches. The details of the EFC bus usage will be described in more detail in subsequent sections. The overall bus structure for the EFC system, though, is illustrated in Figure 3.2.

3.2.2 Memory Arbitration

Since memory accesses are not synchronized, a memory arbiter is needed. It accepts memory requests as inputs and then controls which EFC subsystem may access memory. Because the arbiter is implemented with a PAL, the 186 microprocessor is able to work solely on I/O operations instead of also having to arbitrate the memory bus. The arbiter also prevents the same device from accessing memory with two consecutive requests thus allowing lower priority devices to gain access. In addition to the requests from the PC, 186, DDC, and EFC, it is necessary that the memory be refreshed. This is accomplished by using a counter so that refresh occurs by strobing each row of memory at a rate of 256 refresh cycles every 4 milliseconds.

Besides these requests, there are two more requests that deal more with the control of the memory arbiter. They are the reset and rearbitration signals. Reset is used for, as its name implies, for startup or system reset. After one request has been serviced, the rearbitrate signal is used to honor the next waiting request.

Since there are seven contenders for the memory, three output signals are needed to uniquely identify each contender. The memory arbiter is illustrated in the schematics of Appendix B on page 89.

When multiple requests are active simultaneously, a request is granted ac cording to a predetermined priority scale. The priorities, from highest to lowest, are as follows:

1. DDC (Disk Data Controller) DMA request to move data whenever necessary.
2. Memory refresh.

3. Intel 80186 memory request because it is the system I/O controller.
4. IBM PC-AT DMA request.
5. IBM PC-AT memory request.
6. IBM PC-AT I/O request.
7. EFC DMA request lowest priority since it runs fastest.

All PC requests, however, effectively have the same priority since only one can be serviced at a time. It is unimportant whether a request is for a read or for a write. The arbiter does not differentiate. Its output is used as input to the Memory Sequencer and Write Enabler, and it is these subsystems which together perform either the reading or writing of the data. The Memory Sequencer generates the necessary signals for each device to access memory.

The Write Enabler, whose inputs and outputs are shown on page 90, has as three of its inputs the outputs of the arbiter. The role of the Write Enabler is to set up the system so that words of data are written in a certain manner. The set up may allow for both bytes to be written, only the low byte to be written, or only the high byte. Depending on what the arbiter has selected, only a portion of the Write Enabler inputs are used. Of these inputs, there is one "write" line corresponding to each of the memory users. The one exception is the EFC because it only reads from memory. The remaining inputs, the two groups of A0 and BHE lines, are used by the PC and 186 to selectively write the high and/or low bytes. A0 and BHE correspond to the outputs WEL and WEH, or Write Enable Low and Write Enable High. If WEL is high, then low order bytes are written to memory; similarly for WEH. The disk does not have this option and writes both bytes of a word to the memory.

After the arbiter grants a request and the method of writing, if any is to occur, has been established, the Memory Sequencer is responsible for retrieving or storing the

data in memory. The only inputs to the Memory Sequencer are the Memory Arbiter's outputs. Using the selected request from the arbiter, the sequencer uses the address in the buffer corresponding to the request along with the Row and Column Address Select lines to access the correct address in memory. Then, using the corresponding data buffer, it gets or puts the data there from memory. The Memory Sequencer is shown along with the Memory Arbiter on page 89.

3.2.2.1 PC Memory Requests

The PC is different from the other devices accessing memory in that it may make accesses in three different ways. It can directly access the external (to the PC) EFC memory by using an area in its own upper region of memory which is mapped into the EFC memory. Or, alternatively, a 128K byte window, located at 768K in the PC's memory, may be used to write into one of the four 128K byte areas of the external memory. The third method of addressing memory is to use pseudo-register access to be discussed shortly with the EFC system I/O space.

The PC's address bus is 24 bits wide compared to the EFC memory address bus which is only 18. Thus, the high order bits of addresses off of the PC bus must be cut to only 18 bits. If these high order bits when added to a value determined by the configuration switches equal "11111", then the PC will be directly accessing the external EFC system memory.

To access memory through the window is a little different. When the PC's address is such that it indicates the memory access is through the window, 2 bits from a latch (the Lower Memory Offset Latch) are used as the 2 high order bits of the system memory address. Since the window is only 128K bytes and recall that the system memory is 512K bytes (256K words), then the 2 bits address one of the four 128K blocks in the system memory. The memory addressing schemes are illustrated in Figure 3.3.

3.2.2.2 DMAs in the EFC System

The PC, DDC, and EFC array all use Direct Memory Accesses (DMAs). Much of the supporting DMA hardware for each device is therefore similar. Associated with each device are an 18 bit address counter and a 16 bit data counter. All three also have a 16 bit data buffer for memory to device transfers. The EFC chip will not be writing to the system memory but since the PC and DDC do, each of these two devices has a 16 bit data buffer to transfer data from it to the memory.

To sequence the events of the DMA, each device has an associated finite state machine (FSM). The FSM controls both device-to-memory and memory-to-device transfers. But before any data transfer begins, the 186 microprocessor loads the base address into the address counter and sets a read/write bit depending on which type of DMA will be undertaken. Then, the count down counter is loaded with the byte count, or number of transfers, which must take place. If the DMA will be memory-to-device, the FSM sends a read request to the memory arbiter. Once the request has been honored, the memory sequencer will put the data into the data buffers and notify the device. If, conversely, the DMA will be device-to-memory, the data from the device will be placed into the data buffer, and then a memory write request is sent to the memory arbiter. This continues until all data has been transferred. The PC and PC/DDC interface have data transfer counters operating also so that if the count down counters at each end do not reach zero at the same time, the microprocessor will be interrupted.

3.2.3 The DDC-Disk Interface

The interface between the Disk Data Controller (DDC) and the disk itself can be broken down into two major subsystems: the configuration/status interface and the data interface. The configuration/status interface sends commands to the disk after checking disk status of such things as bits for the number of cylinders and heads, error

status, track offsets, etc. This status data must be sent serially using many handshaking signals. Because status checking or modification is not done too often and because it takes over 12 microseconds to send one bit, these operations will be implemented in software. This significantly reduces the amount of hardware that would otherwise be required.

The other subsystem, the data interface, is more complex. Although most handshaking signals can be connected directly between the disk and DDC, this is not always the case. The disk being used is a soft-sectored ESDI (Enhanced Small Device Interface) drive. A minor problem involves sector pulses. Since there are none on a soft-sectored disk (they would indicate the beginning of a sector), the address mark field is used for this purpose. A more difficult problem to overcome, however, is the handshaking associated with the address mark field. Two signals, Address Mark Enable (AME) and Address Mark Found (AMF) should normally handshake as their names imply. Unfortunately, on an ESDI drive, AME is generated by the DDC only during a disk format. Additional circuitry must be added to produce the AME signal in order that the AMF can be returned to produce a sector pulse.

Because of the complexity of this interface, the reader is referred to Jacob [2], the designer of the DDC-Disk interface, for more detail.

3.2.3.1 EFC System I/O Space

The I/O space of the EFC system is divided into regions accessible only by the PC, only by the 186 microprocessor, and a region accessible by both. The PC accessible I/O space will be discussed first. When an address is put on the System Address Bus by the PC, a decoder determines whether or not the address is in the EFC system's I/O space. The location of this I/O space may be set by the user with the ADDCOMP switches (see pages 80 and 91 in Appendix B). When the switches match the PC's address, an I/O acknowledge signal is sent to a device decoder.

The devices accessible by the PC are memory mapped I/O, the PC Lower Memory Offset Latch, and interrupt registers. The memory mapped I/O appears to the PC as 64 general purpose registers. When these registers have been addressed by the PC, a Memory I/O signal is sent to the PC Memory Address Decode (see page 91 in Appendix B) which generates a PC I/O Request to be sent to the memory arbiter. Since there are 64 registers, only six bits from the system address are used along with the bits set by the user with the ADDCOMP switches. This way, the I/O space may be moved anywhere in memory. These registers are also accessible by the microprocessor, but because it knows of their location in memory, it reads and writes them as normal memory accesses.

The microprocessor accessible I/O works in a little more complex manner. This is partly so because the microprocessor communicates with many more devices and also because the DDC (disk data controller) has registers of its own. In most cases, though, the accesses are straightforward since it is device register addresses that the microprocessor puts on an I/O address bus. A decoder then sends the read or write signal to the appropriate device, and then the data is transferred. The devices in the microprocessor's I/O space which are addressed in this manner include: address and word counters for DMA transfers, the read/write bits for the DDC and PC DMAs, EFC status register (to be discussed in the next chapter), disk command/status interface, and a control signal (for head select, etc.) register for the DDC interface. Except as noted, all of these have been previously discussed. To be discussed shortly and also in the microprocessor's address space are the interrupt system and a FIFO.

The DDC's on-chip registers prevent it from always being this easy though. Since the microprocessor has its own address and data buses, it can access its I/O space while the PC is making other memory accesses in the EFC system. However, should the microprocessor wish to access the DDC, it must wait until the DDC is finished any memory operations because the DDC uses its data bus for both memory and I/O transfers. Therefore if the DDC is in the midst of a memory operation, wait

states must be inserted by the microprocessor into the read/write cycle.

Whenever the PC writes to any memory mapped registers, the lower 5 bits of the address are loaded into a FIFO (see page 93 in Appendix B) at which point the FIFO interrupts the microprocessor indicating that it has data. The micro is then able to read these changed register values. It will only be able to know when 32 of the registers have been addressed though because unfortunately no 6 bit FIFOs are currently available.

3.2.3.2 Interrupts

There are a variety of prioritized interrupts used in the EFC system. They are used to interrupt both the PC and the 186 as one would expect. The FIFO just mentioned in the last section interrupts the 186 with the lowest priority indicating that the memory-mapped registers have been modified. Next in priority is the DDC. It can interrupt the 186 two different ways: when an error occurs while control/status is transferred to or from the disk, or when the DDC asks the 186 to read its error registers. As with the DDC, the EFC and the PC will interrupt when their DMA counters reach zero. These two types of interrupts, however, can be masked by the 186. The 186 can also be interrupted by the PC, and vice versa. For a more thorough discussion, the reader is referred to Jacob [2].

3.3 Summary

The EFC chips described in the first past of this chapter and the rest of the EFC system described in the second part can be viewed as the two major subsystems of a whole EFC system. Each has been designed assuming the existence of the other without knowing the details of operation of the other half. Therefore, the two subsystems cannot be directly "plugged in" to the other. This will be the subject of the next chapter.

Some interfacing circuitry will be needed to coordinate the actions of the two halves to build a functional EFC system.

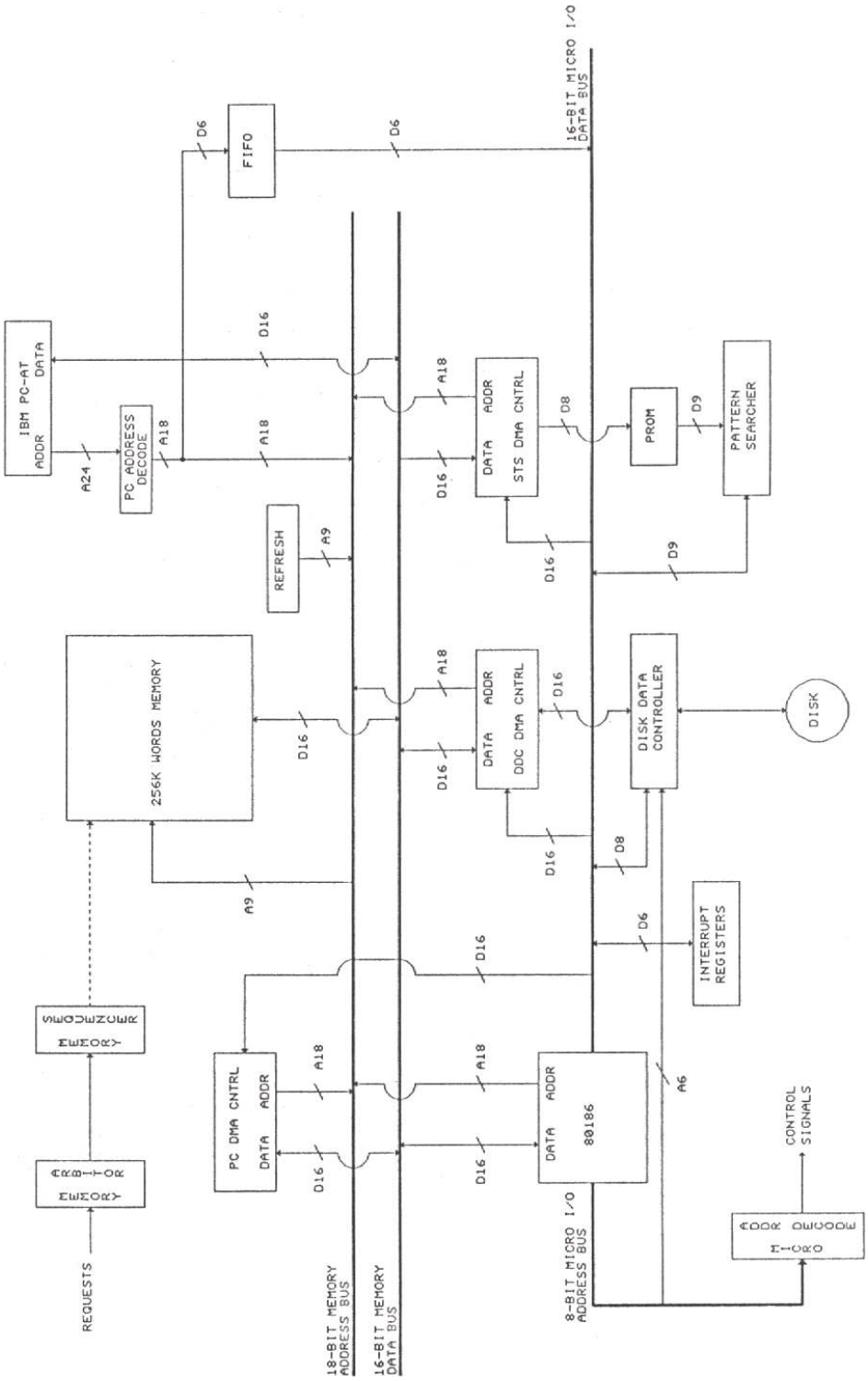

Figure 3.2: EFC System Bus Structure

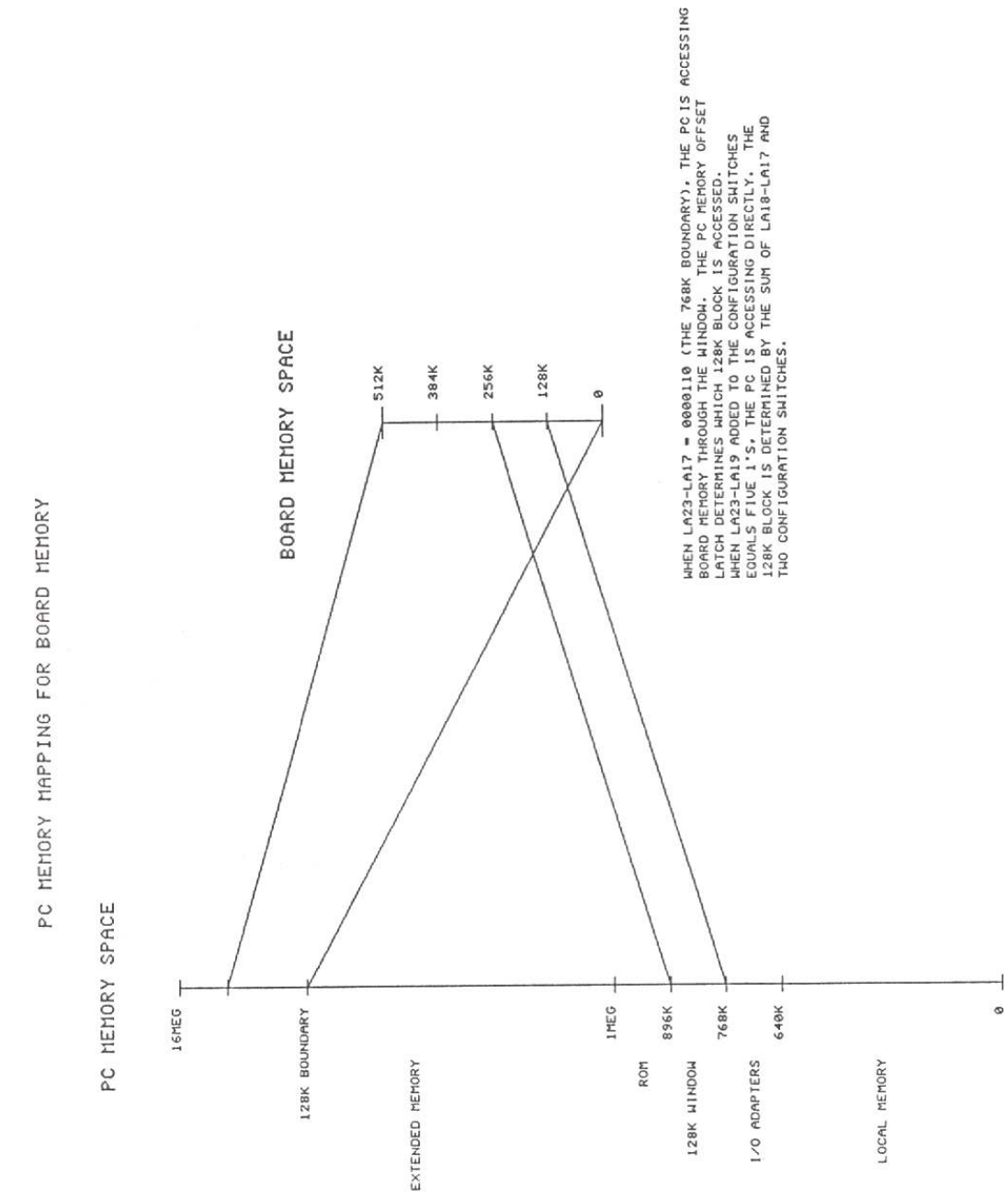

Figure 3.3: PC Memory Mapping for System Memory

CHAPTER 4
THE EFC SYSTEM-CHIP INTERFACE

In conjunction with the EFC chips and the EFC system, an interface between the two must coordinate all data transfers. In the previous chapter it was stated that there are two basic modes in which an EFC chip operates—program and search. The EFC chip, more often than not, will be searching. Because it is capable of operating at much faster data rates than the disk, as much of the memory-to-EFC-chip logic as possible is designed in hardware. EFC programming, on the other hand, involves a small data transfer and is done relatively infrequently so that most of this operation is undertaken in software. At any point in either the program or search mode, there is also a system reset option available to the user.

4.1 EFC Programming

Because much of the programming logic is implemented in software, the circuitry is relatively simple. Before any data transfer begins, the 186 sets a latch in the system which lets the EFC chips know to enter the programming mode of operation. After this, the 186 can begin sending data. The 186 software then will convert all of the data for programming into the 9 bit format the EFC chips require. An 8 bit to 9 bit code converting PROM will not be needed, so only 9 bits of the 16 bit data bus will be used during programming. The sequence of data must be as presented in Tables 3.2 and 3.3 in Chapter 3. When the last of these bytes, has been sent to the EFC chips, the 186 checks the EPC status register bit indicating whether or not programming is complete. When the status register signifies that it is, the 186 resets the program latch. The chips at this point are loaded and searching may begin at any time.

Besides setting the program latch, there are other handshaking signals the EFC chips require. To make the EFC chips remain in the programming mode, the Data Load Reset (**DTLDRST**) signal must be asserted throughout programming starting at least 1 clock cycle before the first bytes arrives and stopping at least 1 clock cycle after the last bytes arrives. At the same time that **DTLDRST** is initially asserted, the Data Delay (**DTDLY**) signal must be asserted for 3 clock cycles. On the second cycle of the **DTDLY**, **VLDT**, or Valid Data signal, will be asserted for exactly 1 cycle. This is the first dummy bytes of the programming data. After **DTDLY** is deasserted, data may be transferred at any time under two constraints: **VLDT** is asserted for exactly one cycle, and data is valid at the rising edge of the (asserted low) **VLDT** signal. This timing sequence, along with the two clocks used by the EFC chips, is shown in Figure 4.1. The figure, for the sake of space, shows the system as it would look if only two bytes of data were needed to fully program it in addition to the dummy data byte. In the actual EFC system, the only difference in timing is that between the two bytes shown would be all the others with associated **VLDT** signals.

4.2 EFC Searching

Speed is more important during searching because of the volume of data used. So rather than have software do the 8 to 9 bit code conversion, a PROM is used to save time. A DMA to the EFC chips, however, puts 16 bits onto the data bus. The PROM, like the EFC chips, accepts one byte at a time. Therefore, a finite state machine (to be described subsequently) passes first the low byte from the DMA and then the high byte. This is made possible by having two transparent 8 bit latches controlled by the state machine between the 16 bit DMA register and the PROM.

The 9 bit data is put onto a data bus which always goes to every EFC chip. By using the connective logic between chips, the data might or might not be used by the chip. Each chip contains only one string matcher so that the only output of a

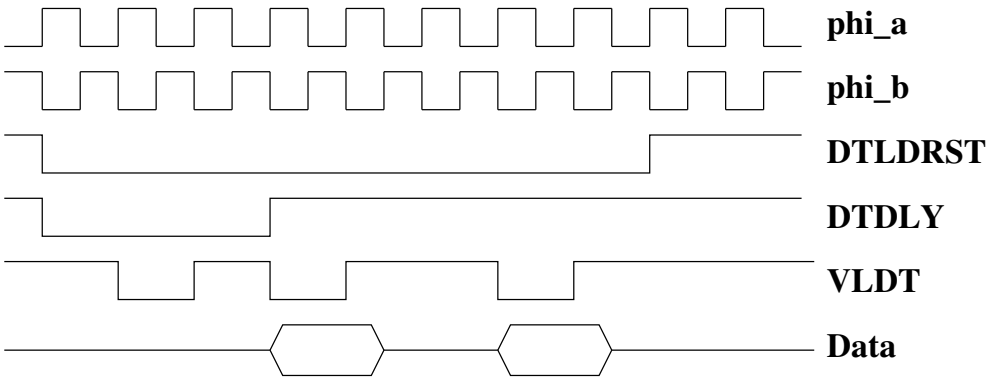

Figure 4.1: EFC Programming Timing Diagram

chip will be the string found signal meaning only that that string has been located not necessarily a matching document. These outputs will be saved in a register so that software may interpret the results in any way it likes (e.g. document found only when all strings are matched or perhaps when just a majority match). But once a document match is made, the document number must be available. This is achieved by attaching a small circuit to the data bus. In it, a FIFO a few bytes, long is constantly shifting data through itself. Whenever the oldest and deepest byte_9 in the FIFO is recognized as a start of document character, the writing of the FIFO is disabled unless another start of document character arrives. In this way, whenever a document match is made, its number will always be available.

The timing of signals for searching turns out to be simpler than for programming. The only signals between the EFC interface and the chips are the Valid Data (**VLDT**) signal again and the bits of data. **VLDT** works the same way as it does for programming. When it is asserted, it must be so for exactly one clock cycle, and the data must be valid at the rising edge of **VLDT**. The timing diagram is shown in Figure 4.2.

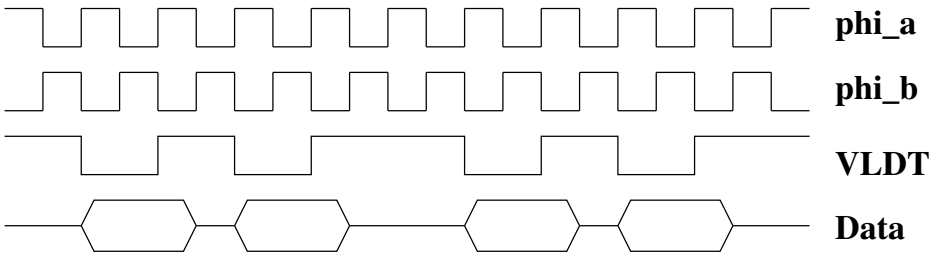

Figure 4.2: EFC Search Timing Diagram

4.3 EFC System-Chip FSM

In the previous sections it was stated that the finite state machine (FSM) for the interface handles the data flow for programming and searching between the EFC system and the EFC chips. This procedure, of course, is more involved than merely pulsing latch output controls on alternating clock cycles. The FSM must also generate and coordinate all handshaking and control signals so that every action is correctly timed. In addition, the two complementary clocks needed by the EFC chips are generated by this FSM. This avoids the problem of skew between control signals and the clocks.

In the system-chip interface, there are two 1 hit latches writable by the mi croprocessor. One latch, when set, means that the system is in the program mode, and when the other is set means that it is in search mode. At its most basic level, the FSM can be thought of as having an idle state, a program branch, and a search branch. Depending on which, if any, of the program and search latches are set, the FSM will proceed through one of the branches. It can never go from one branch to another without first returning to the idle state.

The FSM must be built with one more consideration: there must always be an even number of states. This is because the FSM generates the EFC clocks. There is a type A state where clocks **phi_a** and **/phi_b** are generated. This must *always* be followed by a type B state generating **/phi_a** and **phi_b**. This further implies that

every A state must exit to a B state and every B to an A.

Also, the EFC chips can receive at most one bytes of data every other EFC clock cycle. So four states, or EFC half-cycles, per byte, will be needed. When the EFC is searching, the FSM receives two bytes from the DMA so that at least eight states will be needed in that branch of the state machine. Because speed is so important during searching, it would also be desirable to overlap as much of the data transfer cycle as possible. In Chapter 3 it was mentioned that there are counters keeping track of when a DMA transfer is complete. When this FSM, using the DMA counters, sees that the transfer is not complete, it requests another word of data while it is still giving the EFCs the first of two bytes of data. After both bytes have been transferred, hopefully a new word of data will already be waiting to begin the cycle over again.

The searching branch of the FSM is shown in Figure 4.3. The bold face words are conditions used to decide what state to go to next, and the italicized words are signals asserted in that state. The asserted signals are the same shown in the timing diagrams of the previous sections. The other signals asserted that are not shown on the search timing diagram of Figure 4.2 are used as latch enable and output control signals. These are seen on the schematics of Appendix C. Although the timing diagrams for searching are simpler than those for programming, the search branch of the finite state machine is more complex than the program branch. This is because of the complexity introduced by getting two bytes at a time from a DMA and from overlapping DMA requests.

Programming of the EFC chips is asynchronous, but only one bytes of data at a time will arrive from the micro 186. Since the 186 put the system in the programming mode, the 186 will know that data transfers are necessary. Therefore, the FSM does not make data requests to the 186 but only waits until new data arrives. Timing of data arrival between the FSM and the 186 is not necessary because the 186 operates much more slowly then the FSM. The simpler programming bratch of the FSM is shown

in Figure 4.4. Again, the asserted signals are those from the earlier timing diagrams where the signals not shown on Figure 4.1 are latch output controls, etc.

The EFC chips, with the design of this interface, now are able to communicate with the rest of the EFC system. This is all of the hardware needed to build a functional Electronic Filing Cabinet.

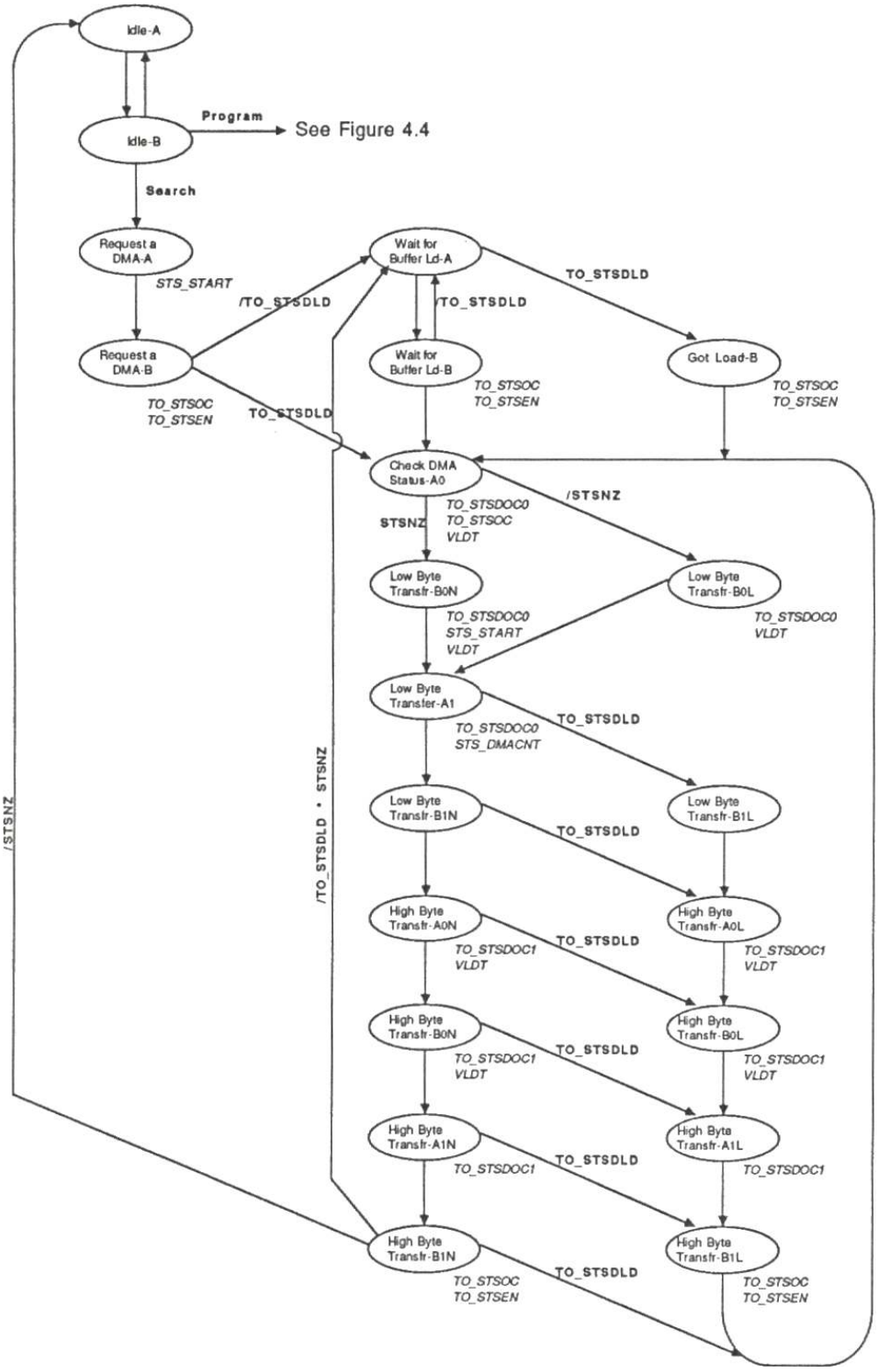

Figure 4.3: FSM Searching Branch

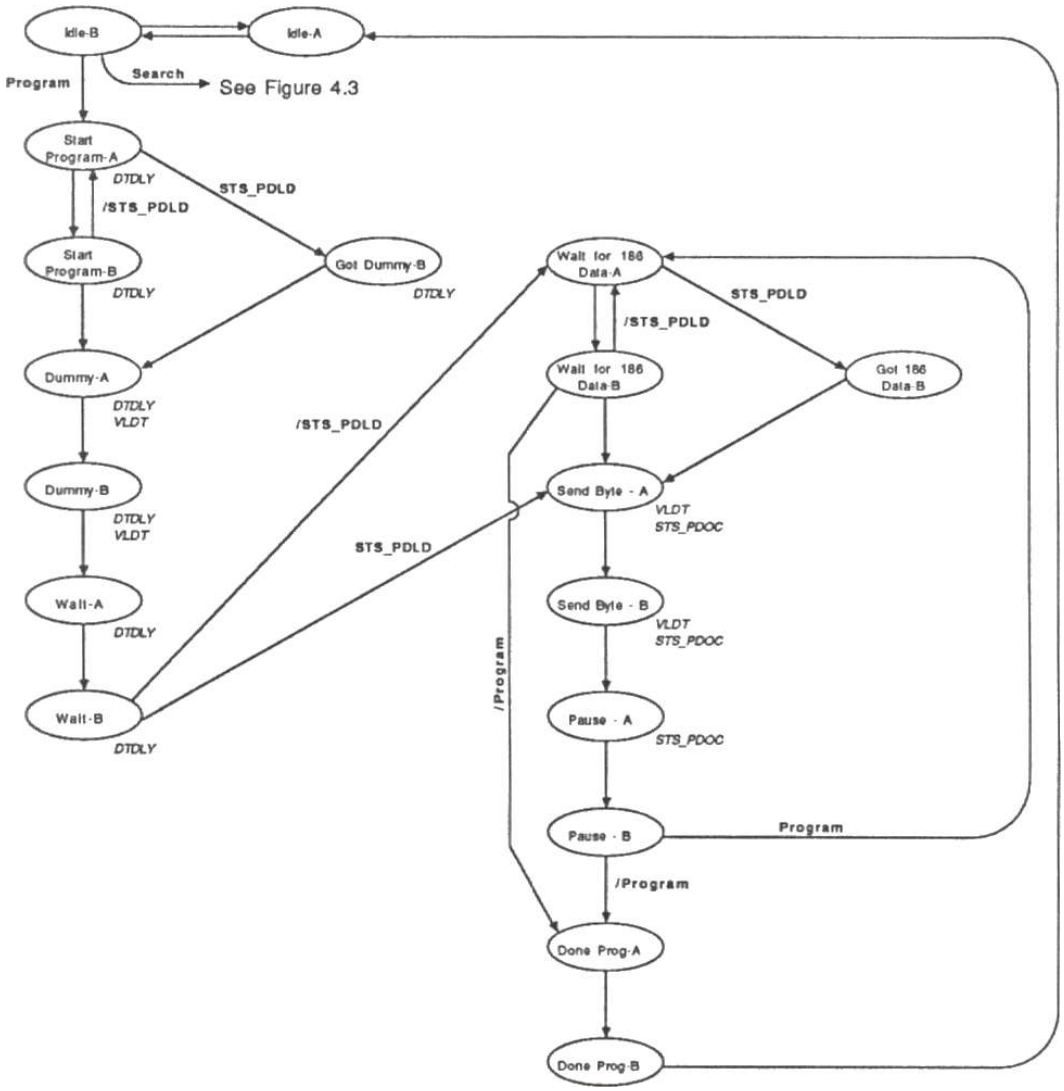

Figure 4.4: FSM Programming Branch

CHAPTER 5
CONCLUSIONS

In this thesis, a first effort at the design of an Electronic Filing Cabinet has been presented. There are surely areas where simplicity or efficiency can be increased, but we have not attempted to design the perfect EFC. Before streamlining the design, it must first be shown that a working design is possible. That is the goal we have achieved.

We are aware, however, of several areas where further improvements could begin. One issue which must be addressed is the modification of the search branch of the interface state machine. After a document match is made, the FSM must loop in wait states until the 186 software has retrieved the document number from the registers holding it. Then the search should continue where it left off.

A substantial undertaking in itself will be the design of a software system which will be easy for office personnel to use while still making full use of everything the EFC system offers. This would perhaps include an option to fix errors in documents matched but with errors in one or more strings. For use by the software, there may also be other useful information to fill have in the EFC status register.

There are undoubtedly other areas where improvement is possible. The system design for a first EFC put forth by Jacob, Shin, Warter, and me is, however, a sound one. The first EFC is scheduled to be built within the next year and should be quite useful in a variety of applications. Chief among its advantages are speed, simplicity of operation, and almost maintenance-free data base. Such a system does not require

the user to have a technical background in order to use its full potential. As a result, systems like this which are capable of handling general office data will allow the user to enjoy the speed and power of a computer helping with routine office tasks.

APPENDIX A

EFC SIMULATOR SOFTWARE

```
#include <stdio.h>
#include <strings.h>
#include "efc_structs.h"

#define The_Stars_Twinkle 1

int     cur_len,          /* length of current word in input stream */
        wd_len,           /* length of current target word */
        tok_high = -1,    /* most tokens used to date */
        new_wd;           /* flag used so that statistics are gathered only
                             on first token follower to match */

struct stats wd_info[20];         /* structure to store EFC statistics */

float wd_occ,                     /* relative occurrence of target word */
      wd_cur_match = 0.0,         /* number of matches against current target */
      matching_tok[5*TOK_NO+1],   /* stats on tok fol activation */
      real_tok[TOK_NO+1];         /* statistics on physical token follower use */

/*************************************************************************

Routine: main

Purpose: Check for proper parameter passing to program and then
         call top-level EFC simulator routine.

Inputs: data base containing words and their relative frequencies of
        occurrence.

Outputs: none.

Other Side Effects: none.

*************************************************************************/
```

```
main(argc, argv)
int argc;
char *argv[];
{
        if (argc != 2)
                error("Usage: efc wordfile", NULL);
        efc(argv[1]);
}

/*************************************************************************

Subroutine: efc

Purpose: Simulate one string matcher of an Electronic Filing Cabinet (EFC)
        system using a data base of words. The data base is formatted in two
        columns where the first is made up of words and the second is made up
        their relative frequencies of occurrence (integers). The sum of these
        frequencies represents the total number of words sampled. For the
        simulation, every word is compared to every other word in the data base.

Inputs: pointer to data base of words.

Outputs: Statistics of the simulation.

*************************************************************************/

efc(fname)
char *fname;
{
        struct str_matcher ss[SS_NO];
        struct state *st_tbl, *make_state_table(), *get_next_state();
        char wd[50];
        int ss_result, i, j, ta, match, newest_tok;
        float occ;
        FILE *fptr, *fopen();
        void rewind(), deactivate_toks();

        for (i = 0; i < 20 ;i++)
                init_stats (&wd_info[i]);
        st_tbl = make_state_table();
        if ((fptr = fopen(fname, "r")) == NULL)
                error("efc: can't open %s", fname);
        while (The_Stars_Twinkle) {
        init_ss_matcher(fname, &ss[0]);
        while (getwd(fptr, wd, &occ, 1) != EOF ) {
```

```
                newest_tok = match = 0;
                new_wd = 1;
                for (i = 0; !match && (i < cur_len); i++) {
                        activate_tok( st_tbl , &(ss[0]) , &newest_tok);
                        for (j = 0; !match && (j < TOK_NO); j++)
                                if (ss [0]. tok[j ]. active) {
                                        ss_result =
                                                ss_match(&ss[0], wd[i],
                                                &(ss [0]. tok[j ]) );
                                        ss [0]. tok[j ]. current_st =
                                                get_next_state(ss [0]. tok[j ]. current_st ,
                                                ss_result );
                                        do_state_actions(&(ss [0]. tok[j ]) );
                                        match = check_tok(&(ss[0].tok[j]) ,
                                                ss [0]. str_len , occ, wd, j);
                                }
                }
                if (match)
                        wd_info[wd_len].mismatches += wd_occ * wd_cur_match;
                if (tok_high >= 0) {
                        wd_info[wd_len].toks_used[tok_high] += occ * wd_occ;
                }
                deactivate_toks(&ss[0]);
        }
        rewind(fptr);
        }
}

/*************************************************************************

Subroutine: getwd

Purpose: Retrieve a word and its relative frequency of occurrence
        from the word data base used in the simulation.

Inputs: fptr    – pointer to word data base.
        buf     – where to store newly read word.
        occ     – frequency of occurrence of new word.
        chg_stats – update frequency of occurrence of either
                    target word or input word.

Outputs: none.

Returns: EOF upon end of data base file.

Other Side Effects: input words are truncated to 16 characters.
```

```
**********************************************************************/

getwd(fptr, buf, occ, chg_stats)
FILE *fptr;
char *buf;
int chg_stats;
float *occ;
{
        int res, len, i;
        res = fscanf(fptr, "%s %f\n", buf, occ);
        if (res != EOF) {
                len = strlen(buf);
                len = (len > 15) ? 15 : len;
                buf[len] = '\0';
                if (chg_stats)
                        cur_len = (float)len;
                else
                        wd_len = (float)len;
        }
        return(res);
}

#include <stdio.h>
#include <strings.h>
#include "efc_structs.h"

extern int tok_high;
extern int wd_len;
extern int new_wd;
extern float wd_cur_match;
extern float wd_occ;
extern float matching_tok[];
extern float real_tok [];
extern struct stats wd_info[];

char sswd[50];

/**********************************************************************

Subroutine:  init_stats

Purpose: Initialize  global  data  structure  to  start-up values.  The
        structures  are used to gather  statistics  on the functioning  of the
        EFC simulations.
```

Inputs: pointer to such a data structure.

Outputs: none.

Other Side Effects: none.

**********/

```
void init_stats ( st_str )
struct stats *st_str ;
{
        int i;

        st_str->mismatches = 0.0;
        st_str->subtot = 0;
        for (i = 0; i < TOK_NO; i++) {
                real_tok[i+1] = 0.0;
                st_str->toks_used[i] = 0;
        }
        for (i=0; i < 5*TOK_NO; i++)
                matching_tok[i+1] = 0.0;
        for (i=0; i < 4; i++)
                st_str->errs[i] = 0;
}
```

/**********

Subroutine: error

Purpose: Quick exit from program upon irrecoverable error condition.

Inputs: s1 can be a string constant or a printf formatting string.
s2 must be a string constant.

Outputs: none.

Other Side Effects: Program execution halted with no data saved.

**********/

```
void error(s1, s2)
char *s1, *s2;
{
        fprintf(stderr, s1, s2);
        fprintf(stderr, "\n");
        exit(1);
}
```

```
/*************************************************************************

Subroutine: salloc

Purpose: To allocate memory to be used for representation of a state
        in the EFC error tolerance state machine.

Inputs: none.

Outputs: none.

Returns: pointer to newly allocated memory.

Other Side Effects: none.

*************************************************************************/

struct state *salloc ()
{

        char *malloc();

        return((struct state *) malloc(sizeof(struct state)));
}

/*************************************************************************

Subroutine: nalloc

Purpose: To allocate memory to be used for representation of state
     transitions in the EFC error tolerance state machine.

Inputs: none.

Outputs: none.

Returns: pointer to newly allocated memory.

Other Side Effects: none.

*************************************************************************/

struct new_state *nalloc()
{
        char *malloc();
```

```
        return((struct new_state *) malloc(sizeof(struct new_state)));
}

struct new_state *init_new_state(trans, sptr, tss)
int trans;
struct state *sptr;
struct new_state *tss;
{
        struct new_state *ns_ptr;

        ns_ptr = nalloc();
        ns_ptr->transition = trans;
        ns_ptr->next_state = sptr;
        ns_ptr->next = tss;
        return(ns_ptr);
}

/*****************************************************************

Subroutine:  init_state

Purpose:  Initialize  a new_state being added to the  state  machine.

Inputs:  no       - this  state's number in the state machine.
         actions  - binary pattern representing  state  actions  to  be taken.
         sts      - points to another state in  state  machine state table.

Outputs: none.

Other Side Effects:  memory is allocated for  the  state.

Returns: pointer  to  newly created and  initialized   state.

*****************************************************************/

struct state * init_state (no, actions, sts)
int no;
int actions;
struct state *sts;
{
        struct state *st_ptr;

        st_ptr = salloc();
        st_ptr->state_no = no;
        st_ptr->actions = actions;
```

```
        st_ptr->trans_lst = NULL;
        st_ptr->next = sts;
        return(st_ptr);
}

/*************************************************************************

Subroutine: make_state_table

Purpose: Generate the EFC error tolerance state machine representation.

Inputs: none.

Outputs: none.

Returns: pointer to the state machine table.

Other Side Effects: memory allocated for table generation.

*************************************************************************/

struct state *make_state_table()
{
        struct state *tmps[8];
        struct new_state *tmpns, *tmpns2, *common;

        /* create the seven states */
        tmps[7] = init_state (7, 012, NULL);
        tmps[6] = init_state (6, 010, tmps[7]);
        tmps[5] = init_state (5, 06, tmps[6]);
        tmps[4] = init_state (4, 026, tmps[5]);
        tmps[3] = init_state (3, 0, tmps[4]);
        tmps[2] = init_state (2, 01, tmps[3]);
        tmps[1] = init_state (1, 040, tmps[2]);

        /*create fransition list for each state */
        tmpns  = init_new_state(034, tmps[6], NULL);
        tmpns2 = init_new_state(014, tmps[4], tmpns);
        tmpns  = init_new_state(010, tmps[4], tmpns2);
        tmpns2 = init_new_state(04,  tmps[1], tmpns);
        tmpns  = init_new_state(0,   tmps[1], tmpns2);

        tmps[7]->trans_lst = tmpns;
        tmps[6]->trans_lst = tmpns;
        tmps[5]->trans_lst = tmpns;
        tmps[3]->trans_lst = tmpns;
```

```
        tmps[2]->trans_lst = tmpns;

        tmpns  = init_new_state(035, tmps[6], NULL);
        tmpns2 = init_new_state(011, tmps[5], tmpns);
        tmpns  = init_new_state(015, tmps[3], tmpns2);
        tmpns2 = init_new_state(05,  tmps[3], tmpns);
        tmpns  = init_new_state(01,  tmps[1], tmpns2);
        tmps[4]->trans_lst = tmpns;

        tmpns  = init_new_state(036, tmps[7], NULL);
        tmpns2 = init_new_state(016, tmps[5], tmpns);
        tmpns  = init_new_state(012, tmps[5], tmpns2);
        tmpns2 = init_new_state(06,  tmps[2], tmpns);
        tmpns  = init_new_state(02,  tmps[2], tmpns2);
        tmps[1]->trans_lst = tmpns;

        return(tmps[1]);
}

/*************************************************************************

Subroutine:  get_next_state

Purpose: Given the current inputs and current, transit to the next state.

Inputs: cur_st  - pointer into state table representing current_state.
        input   - character matcher outputs during current_state.

Outputs: none.

Returns: pointer into state table to next state.

Other Side Effects: none.

*************************************************************************/

struct state *get_next_state(cur_st, input)
struct state *cur_st;    /* current_state number */
int input;               /* current input from character matcher */
{
        struct state *ps;
        struct new_state *pns;

        pns = cur_st->trans_lst;
        input = input & 020 ? input | 034 : input;       /* mask don't cares */
        while (pns->transition != input && pns != NULL)
```

```
            pns = pns->next;
        if (pns == NULL)
            error("Unknown token transition %d", input);
        return(pns->next_state);
}

/*********************************************************************

Subroutine: do_state_actions

Purpose: Execute actions for current_state as stored in state machine table.

Inputs: pointer to an active token follower.

Outputs: none.

Other Side Effects: the error flags, error counter, and token pointer ( ss_cell ) in the
    token follower may be updated depending on the state actions performed.

*********************************************************************/

void do_state_actions(tok)
struct tok_follow *tok; /* token follower moving through state machine */
{
        struct state *ps;
        struct new_state *pns;

        tok->ref = (tok->current_st->actions & 040) ? 1 : 0;
        tok->ef = (tok->current_st->actions & 020) ? 1 : 0;
        if (tok->current_st->actions & 010)
            tok->ss_cell++;
        if (tok->current_st->actions & 04)
            tok->ss_cell += 2;
        if (tok->current_st->actions & 02)
            tok->err_cnt++;
        tok->abort = (tok->current_st->actions & 01) ? 1 : 0;
}

/*********************************************************************

Subroutine: init_ch_matcher

Purpose: Initialize a character matcher with a character group.

Inputs: em chlst pointer to character matcher to be initialized .
```

```
−character group to initialize matcher with.

Outputs: none.

Other Side Effects: none.

**********************************************************************/

void init_ch_matcher(cm, chlst)
struct ch_matcher *cm;
char *chlst;
{
        cm->ch_lst = chlst;
}

/**********************************************************************

Subroutine: init_ss_matcher

Purpose: Initialize string matcher with token or string to be searched for.
        This token is gotten from a data base of target words.

Inputs: fname  − pointer to data base of words to search for.
        sm     − pointer to string matcher to be initialized .

Outputs: none.

Other Side Effects: all token followers in string matcher deactivated,
        pointer into target word data base incremented,
        program execution halted when end of data base is reached.

**********************************************************************/

void init_ss_matcher(fname, sm) /* initialize substring matcher */
char *fname;
struct str_matcher *sm;
{
        static char chlst [16][50];
        static short int first = 1;
        char wd[50];
        int i, j;
        FILE *fopen();
        static FILE *dbptr;
        void print_stats(), deactivate_toks();
        char *strcpy();
```

```
        if ( first ) {
                first  = 0;
                if ((dbptr= fopen(fname, "r")) == NULL)
                        error("init_ss_matcher: can't open %s", fname);
        }
        j = getwd(dbptr, wd, &wd_occ, 0);
/* strcpy(sswd, wd);*/
        if (j == EOF) {
                fclose(dbptr);
                print_stats();
                error("\nFinished.", NULL);
        }
        wd_info[wd_len].mismatches -= wd_occ * wd_occ;
        deactivate_toks(sm);
        sm->str_len = strlen(wd) - 1;
        sm->enabled = 1;
        for (i = 0; i <= sm->str_len; i++) {
                chlst[i][0]  = wd[i];
                chlst[i][1]  = '\0';
                init_ch_matcher(&(sm->reg_matcher[i]), chlst[i]);
        }
}

/*************************************************************************

Subroutine: ch_match

Purpose: Match an incoming character with what is loaded in character matcher.

Inputs: cm      - pointer to a character matcher.
        c       - character to match against what is stored in matcher.

Outputs: none.

Returns: 1 if match is made, 0 if not.

*************************************************************************/

int ch_match(cm, c)
struct ch_matcher *cm;
char c;
{
        int i, found;

        found = 0;
/
```

```
        for (i = 0; i < strlen(cm->ch_lst) && !Jound; i++)
                if (cm->ch_lst[i] == c)
                        found = 1;
/

        if (cm->ch_lst[0] == c)
                found = 1;

        return(found);
}

/*********************************************************************

Subroutine: ss_match

Purpose: Look at three character matcher outputs for the expected character,
        the next expected chatacter, and the second expected character.

Inputs: sm      - pointer to a string matcher.
        c       - current incoming character in input stream.
        tok     - pointer to one of the string matcher's active token followers.

Outputs: none.

Returns: integer where lower order three bits represent, respectively, the
        character matcher outputs for second previous, current, and next
        expected chatacters.

Other Side Effects: none.

*********************************************************************/

int ss_match(sm, c, tok)
struct str_matcher *sm; /* a substring matcher */
char c;                 /* current character in input stream */
struct tok_follow *tok; /* token follower */
{
        int val;

        val = 0;

        if (ch_match(&(sm->reg_matcher[tok->ss_cell]), c ) == 1)
                val = val | 020;
        if (tok->ss_cell < (sm->str_len) &&
                (ch_match(&(sm->reg_matcher[(tok->ss_cell)+1]), c) == 1))
                val = val | 010;
```

```
        if (tok->ss_cell > 1 &&
                (ch_match(&(sm->reg_matcher[(tok->ss_cell)-2]), c) == 1))
                val = val | 04;
        if (tok->ref == 1)
                val = val | 02;
        if (tok->ef == 1)
                val = val | 01;
        return(val);
}

/*********************************************************************

Subroutine: activate_tok

Purpose: Try to activate a new token follower for a newly arrived
        character in the imput stream.

Inputs: st_tbl  - pointer to the error tolerance state machine table.
        ss      - pointer to a string matcher.
        newest_tok -    integer representing what order this token follower
                        is being activated in th sequence of token followers
                        trying to match the current incoming word with the
                        target.

Outputs: newest_tok is incremented only if a token follower is available
        for activation.

Returns: 1 if a token follower is activated, 0 if not.

Other Side Effects: none.

*********************************************************************/

int activate_tok(st_tbl, ss, newest_tok)
struct state *st_tbl;
struct str_matcher *ss;
int *newest_tok;
{
        int i;

        i = 0;
        while (ss->tok[i].active == 1 && i < TOK_NO)
                i++;
        if (i != TOK_NO) {
                if (i > tok_high)
                        tok_high = i;
```

```
                ss->tok[i].current_st = st_tbl->next->next;
                ss->tok[i].err_cnt = 0;
                ss->tok[i]. ss_cell = 0;
                ss->tok[i].ref = 0;
                ss->tok[i].ef = 0;
                ss->tok[i].abort = 0;
                ss->tok[i].active = 1;
                ss->tok[i].number = ++(*newest_tok);
        }
        return((i == TOK_NO) ? 0: 1);
}

/*************************************************************************

Subroutine: deactivate_toks

Purpose: Deactivate all token followers in a string matcher.

Inputs: pointer to a string matcher.

Outputs: none.

Returns: none.

Other Side Effects: none.

*************************************************************************/

void deactivate_toks(ss)
struct str_matcher *ss;
{
        int i;

        tok_high = -1;
        wd_cur_match = 0;
        for (i = 0; i < TOK_NO; i++)
                ss->tok[i].active = 0;
}

/*************************************************************************

Subroutine: check_tok

Purpose: Check to see if token follower should be deactivated because
        either a match or abort condition has occurred. If so, gather
        appropriate statistics .
```

```
Inputs: tok     – pointer to a token follower.
        len     – length of the target string being searched for.
        occ     – relative frequency of occurrence of current word in
                        input stream.
        wd      – current word in imput stream.
        tok_ind – number of physical token follower (not activation order).

Outputs: none.

Returns: 1 if a match condition exists, 0 if not.

Other Side Effects: Abort signal in token follower may be set if number
        of errors encountered exceeds maximum nummber allowed. Token follower
        will be deactivated if a match or abort condition exists.

**********************************************************************/

int check_tok(tok, len, occ, wd, tok_ind)
struct tok_follow *tok;
int len;
float occ;
char *wd;
int tok_ind;
{
        int res;

        res = 0;
        if (len <= 6 && tok->err_cnt == 1)
                tok->abort = 1;
        else if ((len == 7 || len == 8) && tok->err_cnt == 2)
                tok->abort = 1;
        else if ((len == 9 || len == 10) && tok->err_cnt == 3)
                tok->abort = 1;
        else if (tok->err_cnt == 4)
                tok->abort = 1;
        else if (tok->ss_cell == 15 || tok->ss_cell >= len+1) {
                if (new_wd)
                        wd_cur_match += occ;
                res = 1 ;
/*printf("target '%s' matched %s\n", sswd, wd);*/
                wd_info[wd_len].errs[tok->err_cnt] += wd_occ * occ;
                matching_tok[tok->number] += wd_occ * occ;
                real_tok[tok_ind+1] += wd_occ * occ;
                tok->active = 0;
                new_wd = 0;
```

```
        }
        if (tok->abort == 1) /* abort search */
                tok->active = 0;
        return(res);
}

/*************************************************************************

Subroutine: print_stats

Purpose: Make final computations on statistics gathered in many of the
        subroutines and print the results.

Inputs: none.

Outputs: none.

Returns: none.

Other Side Effects: All global data structures used in gathering
        statistics are altered to put data in its final form.

*************************************************************************/

void print_stats()
{
        float mm_tot, avg_wd, avg_tot;
        float tmp_tottoks, tmp_wdtot, tokerrs[4], allerrs ;
        float toks_subtot[TOK_NO], toks_tot, wdtot, match_tok_tot, real_tok_tot;
        int i, j, mst_tok;

        real_tok_tot = match_tok_tot = 0.0;
        wdtot = toks_tot = allerrs = mm_tot = avg_tot = 0.0;
        for (j = 0; j < TOK_NO; j++)
                toks_subtot[j] = 0.0;
        for (j = 0; j < 4; j++)
                tokerrs[j] = 0.0;
        mst_tok = 0;
        for (i = 3; i < 17; i++) {
                tmp_tottoks = tmp_wdtot = avg_wd = 0.0;
                wd_info[i].mismatches /= WDS;
                wd_info[i].mismatches /= WDS;
                printf("\n%d letter target string:\n\n", i);
                if (wd_info[i].mismatches)
                        printf("  %1.8f mismatches.\n\n",
                                wd_info[i].mismatches);
```

```
        mm_tot += wd_info[i].mismatches;
for(j = 0; j < TOK_NO; j++)
        if (wd_info[i].toks_used[j]) {
                avg_wd += wd_info[i].toks_used[j] * (j+1);
                avg_tot += wd_info[i].toks_used[j] * (j+1);
                toks_subtot[j] += wd_info[i].toks_used[j];
                toks_tot += wd_info[i].toks_used[j];
                tmp_wdtot += wd_info[i].toks_used[j];
                wdtot += wd_info[i].toks_used[j];
        }
for (j = 0; j < TOK_NO; j++)
        if (wd_info[i].toks_used[j]) {
                if (j > mst_tok)
                        mst_tok = j;
                printf("  %1.8f used %d tokens.\n",
                        wd_info[i].toks_used[j]/tmp_wdtot, j+1);
        }
for(j = 0; j < 4; j++) {
        tmp_tottoks += wd_info[i].errs[j];
        tokerrs[j] += wd_info[i].errs[j];
        allerrs += wd_info[i].errs[j];
}
printf("\n");
for (j = 0; j < 4; j++)
        if (wd_info[i].errs[j])
                printf("  %1.8f of matching tokens had %d errors.\n",
                        wd_info[i].errs[j]/tmp_tottoks, j);
if (avg_wd)
        printf("\n Average: %1.8f tokens per %d-letter word.\n",
                avg_wd/tmp_wdtot, i);
}
printf("\n%1.8f total mismatches.\n\n", mm_tot);
for(j=0; j < TOK_NO; j++)
        printf("%1.8f used %d tokens.\n", toks_subtot[j]/toks_tot, j+1);
printf("\nAverage: %1.8f tokens per word.\n", avg_tot/wdtot);
printf("Most:    %d tokens.\n\n", mst_tok+1);

for (j = 0; j < 4; j++)
        printf("%1.8f had %d errors.\n", tokerrs[j]/allerrs, j);
for (j = 1; j <= 5*TOK_NO; j++)
        match_tok_tot += matching_tok[j];
for (j = 1; j <= 5*TOK_NO; j++)
        matching_tok[j] /= match_tok_tot;
for (j = 1; j <= TOK_NO; j++)
        real_tok_tot += real_tok[j];
for (j = 1; j <= TOK_NO; j++)
```

```
                real_tok[j] /= real_tok_tot;
        printf("\n");
        for(j = 1; j < 5*TOK_NO+1; j++)
                if (matching_tok[j])
                        printf("%1.8f words were matched with activated token %d\
    n",
                                matching_tok[j], j);
        printf("\n");
        for (j = 1; j<= TOK_NO; j++)
                if (real_tok[j])
                        printf("%1.8f words were matched with the real token %d\n
    ",
                                real_tok[j], j);
}
```

APPENDIX B

EFC SYSTEM CIRCUIT SCHEMATICS

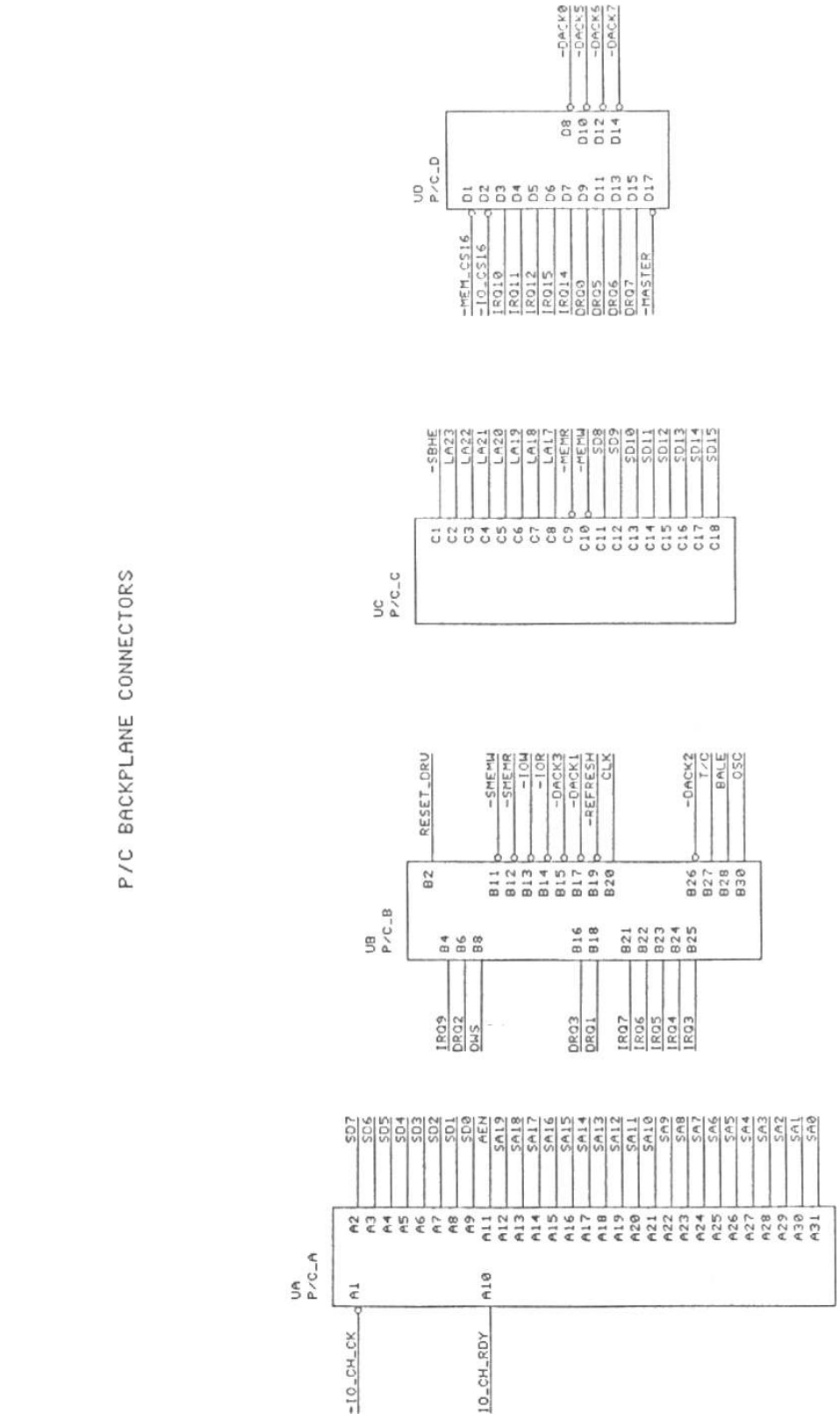

Figure B.1: PC Backplane Connectors

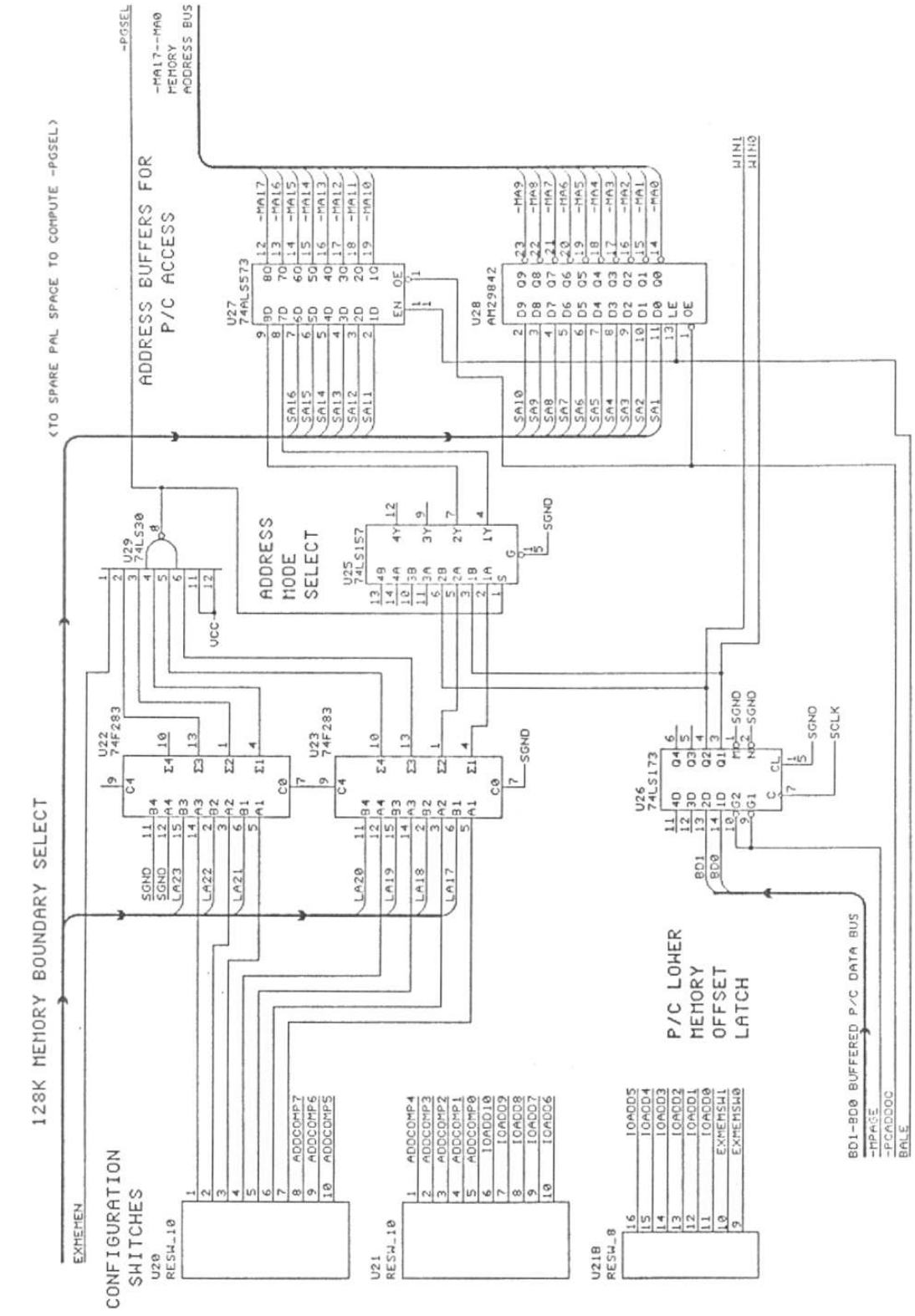

Figure B.2: PC Memory Address Select and Buffers

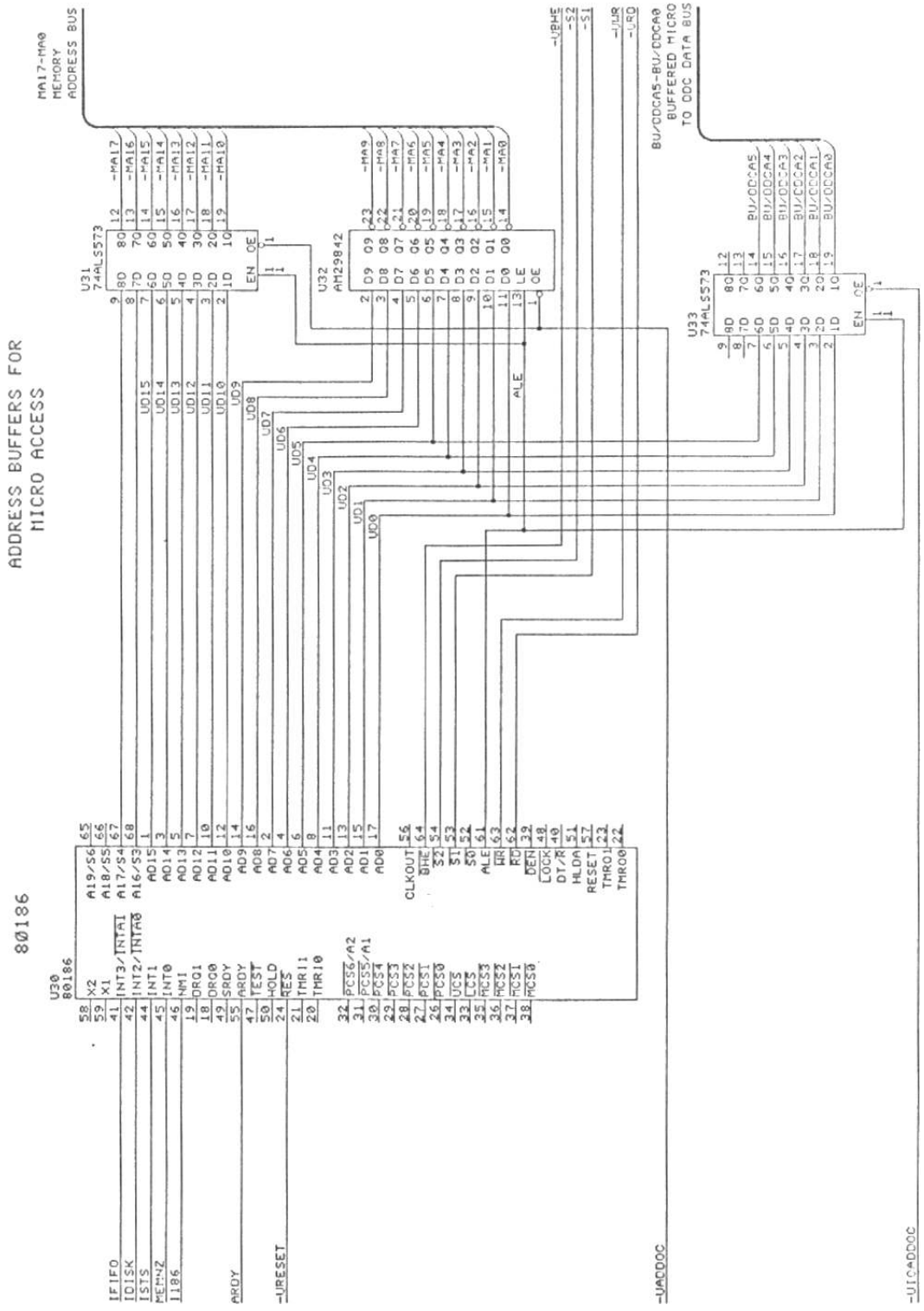

Figure B.3: 80186 and Address Buffers

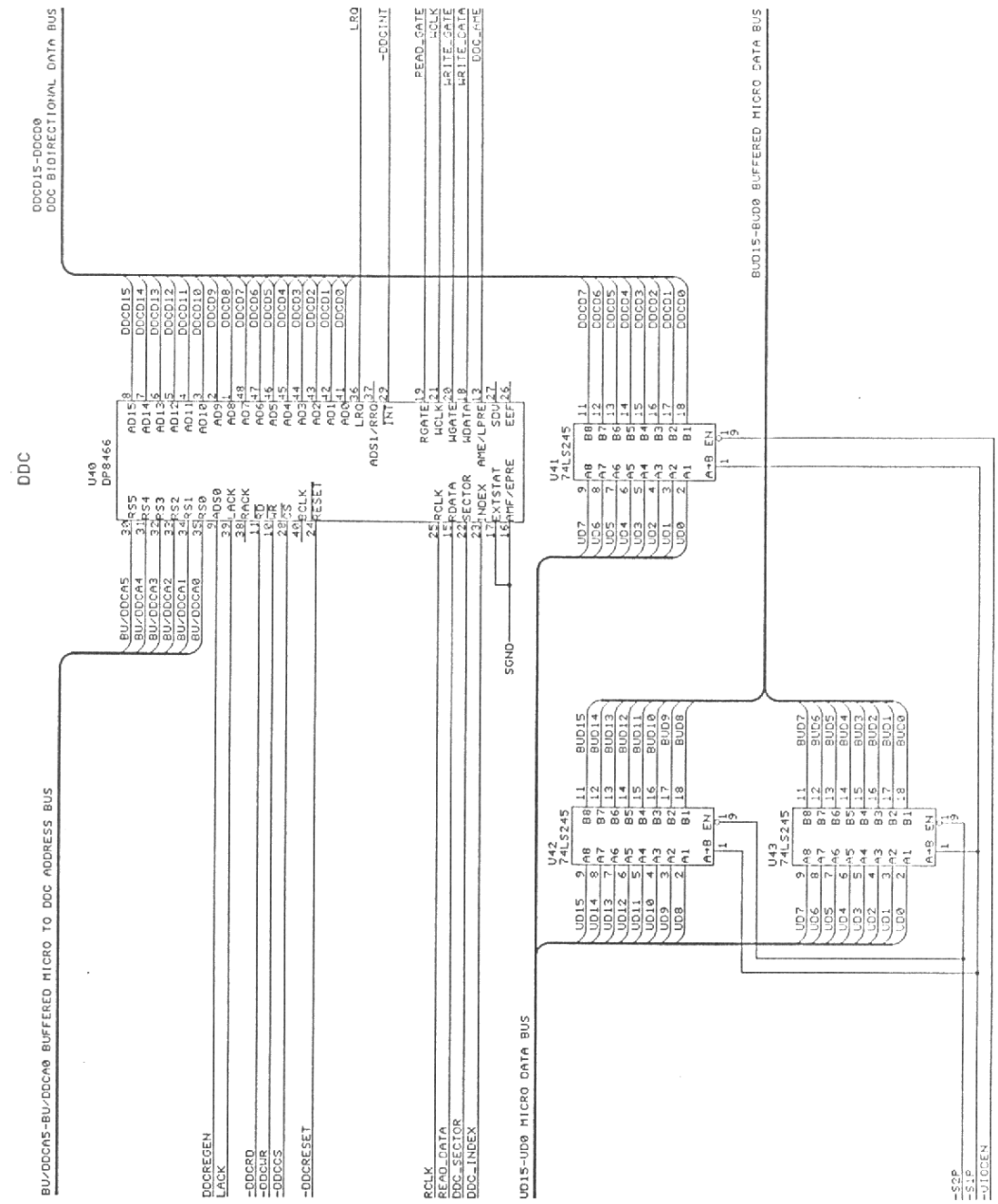

Figure B.4: Micro I/O Data Buffers and Disk Data Controller (DDC)

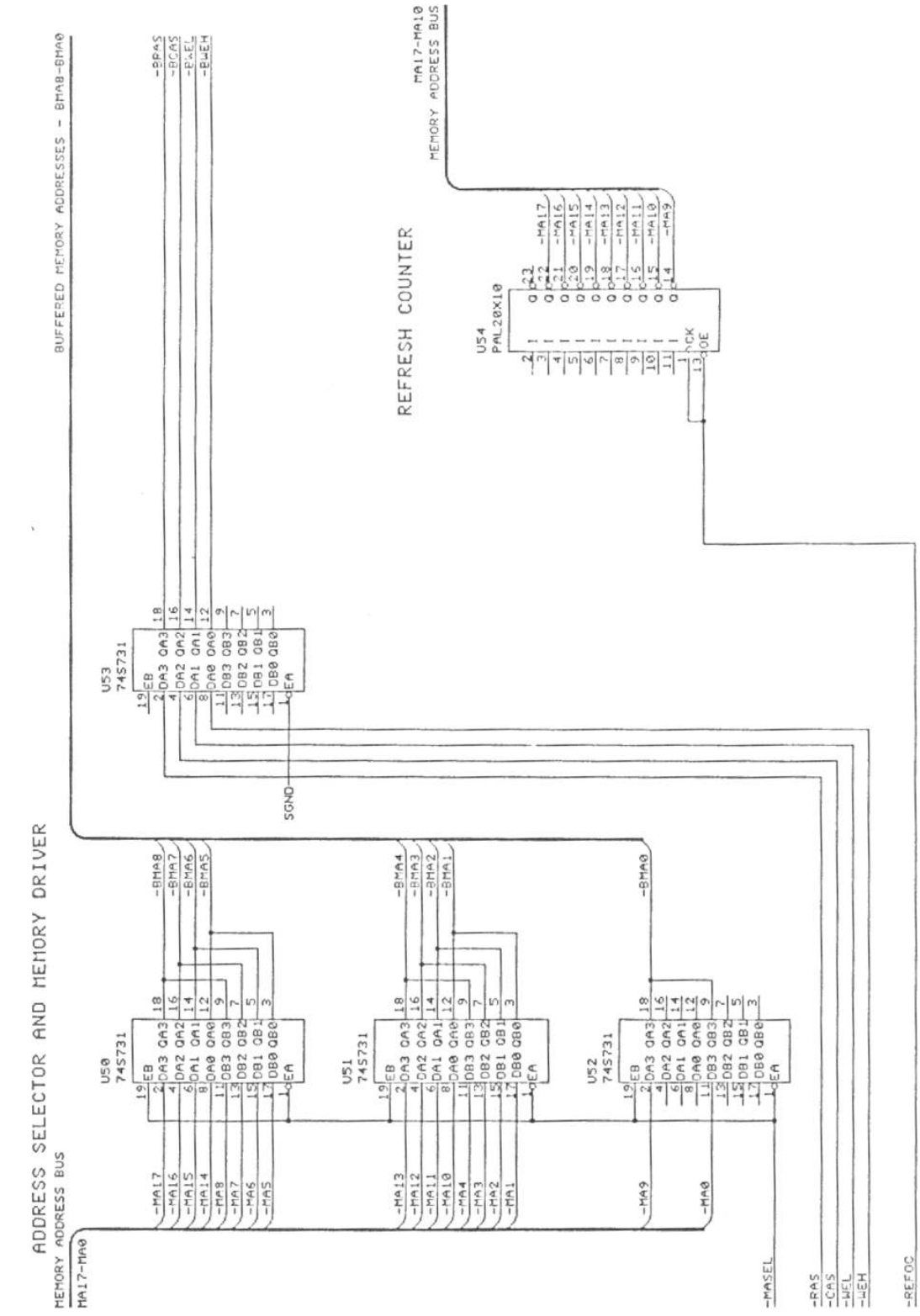

Figure B.5: Address Selector and Memory Driver and Refresh Counter

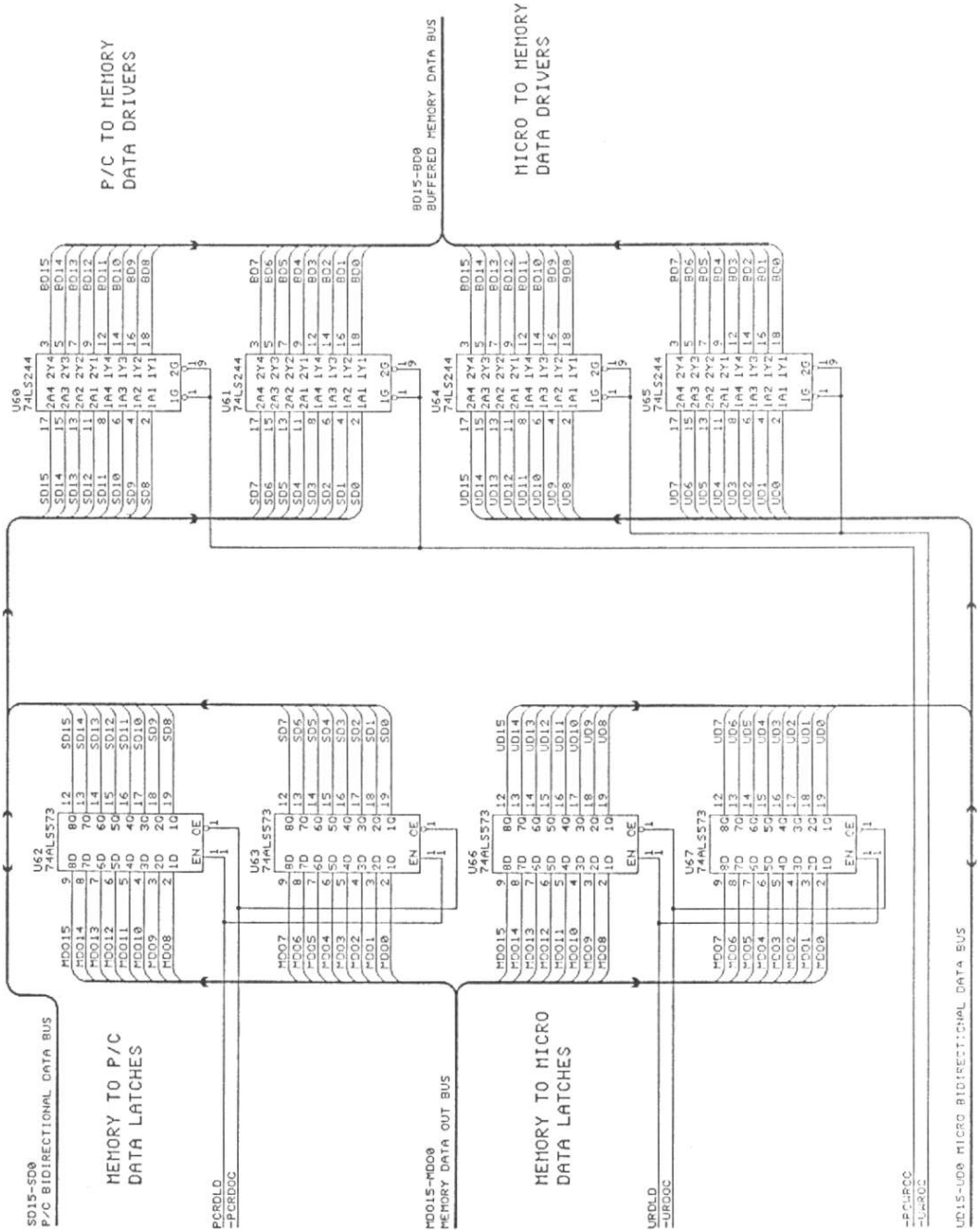

Figure B.6: PC-Memory and Micro-Memory Data Latches and Drivers

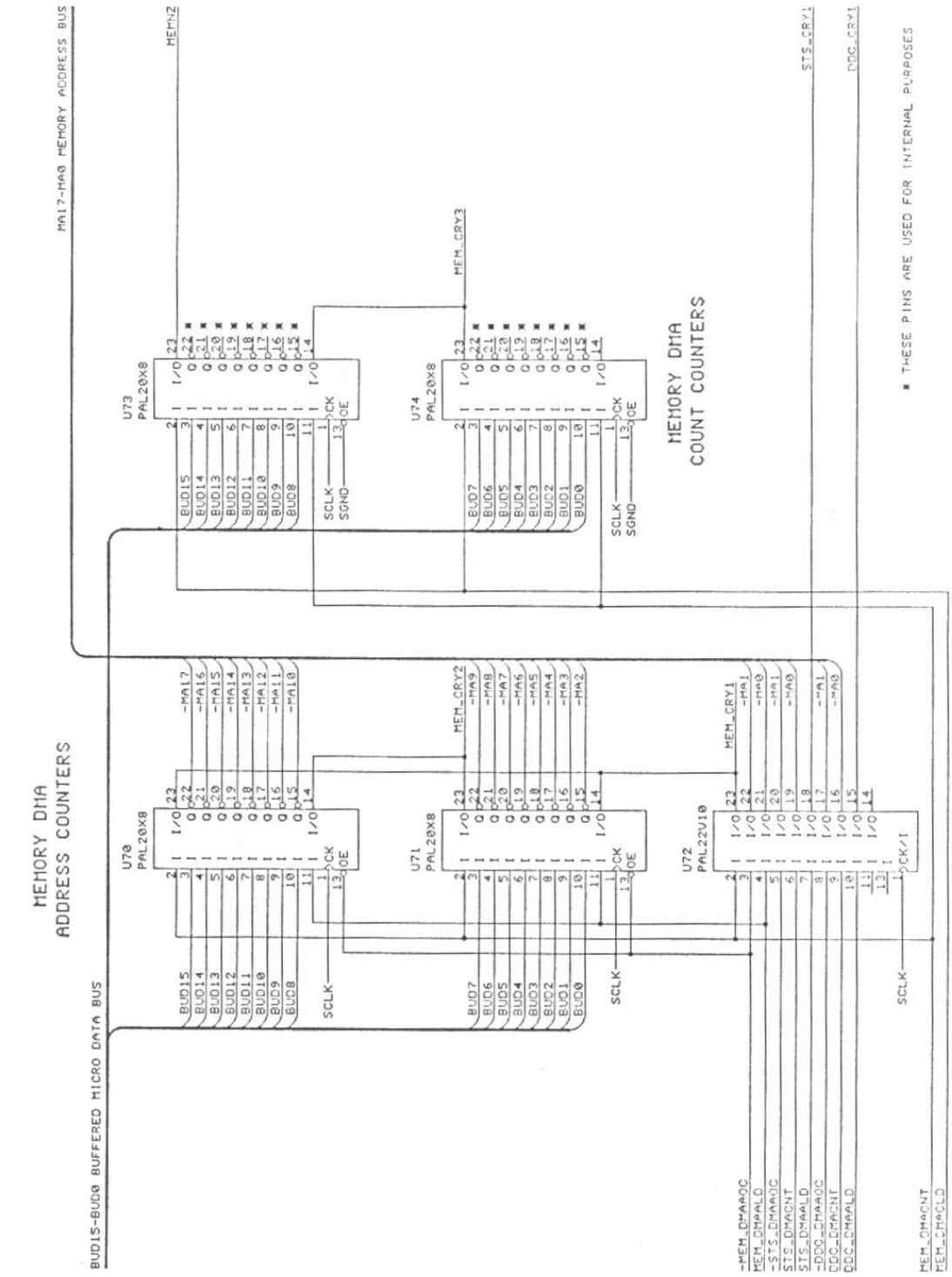

Figure B.7: PC DMA Address and Count Counters

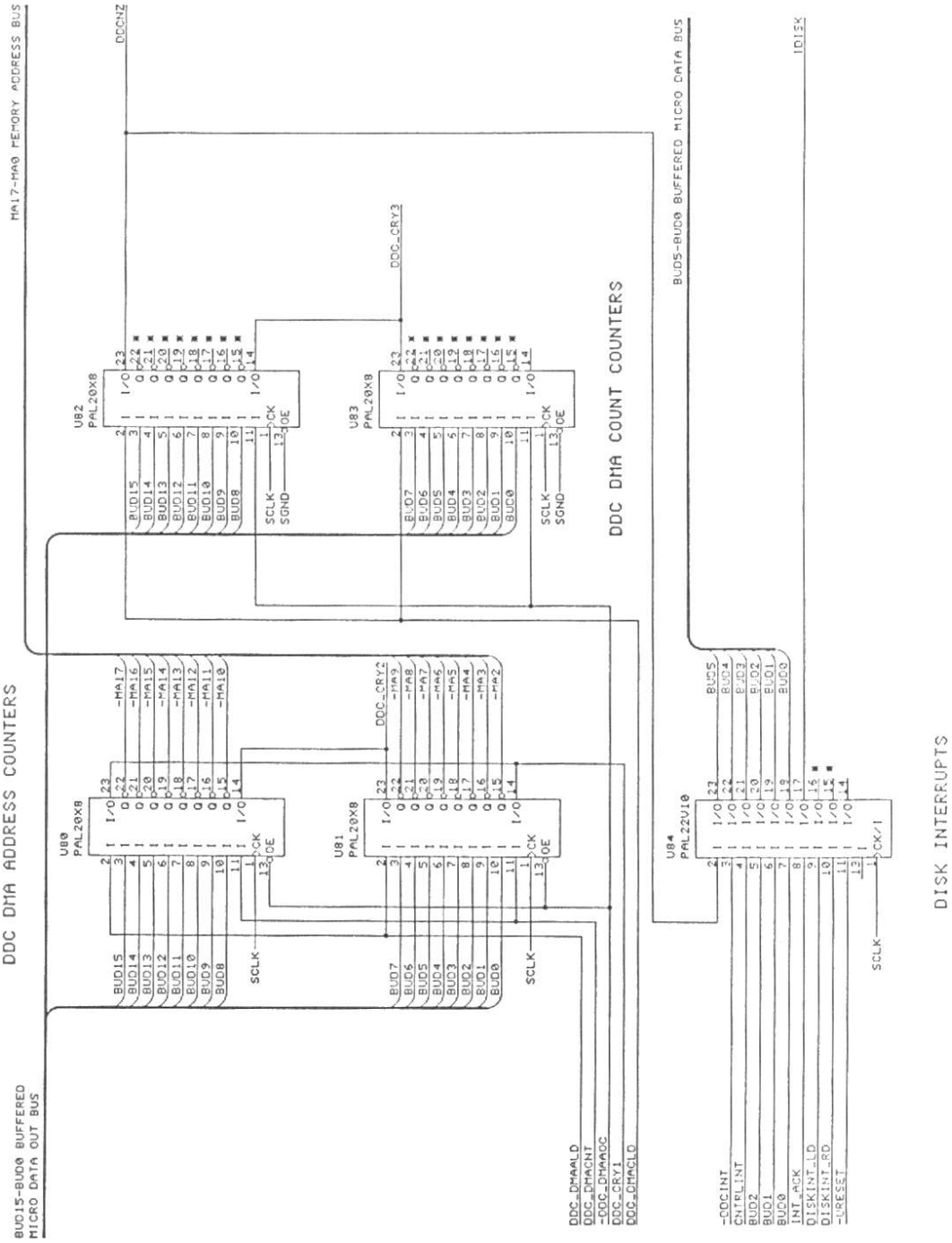

Figure B.8: DDC DMA Address and Count Counters and Disk Interrupt Register

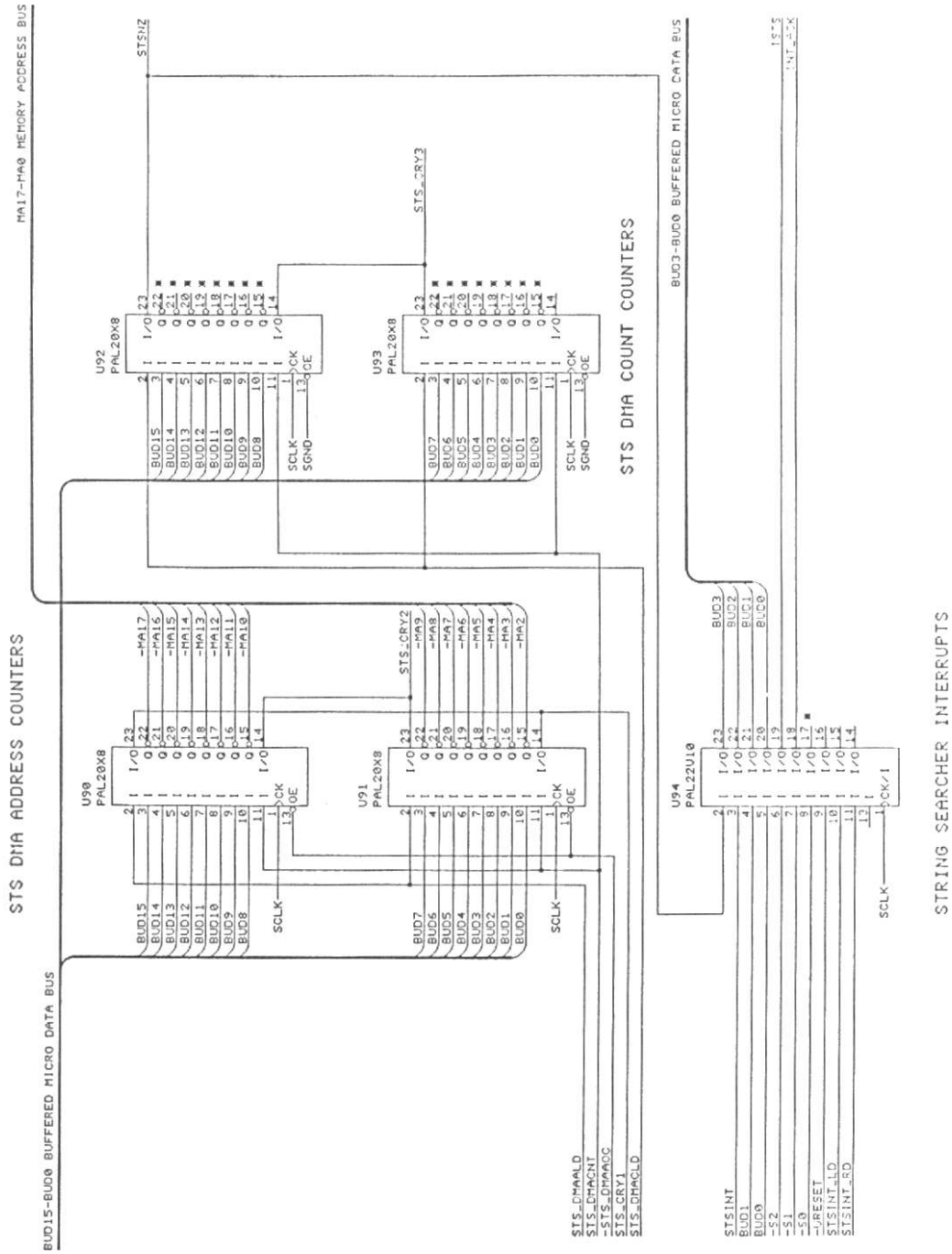

Figure B.9: String Searcher (EFC) Address & Count Counters and Interrupt Register

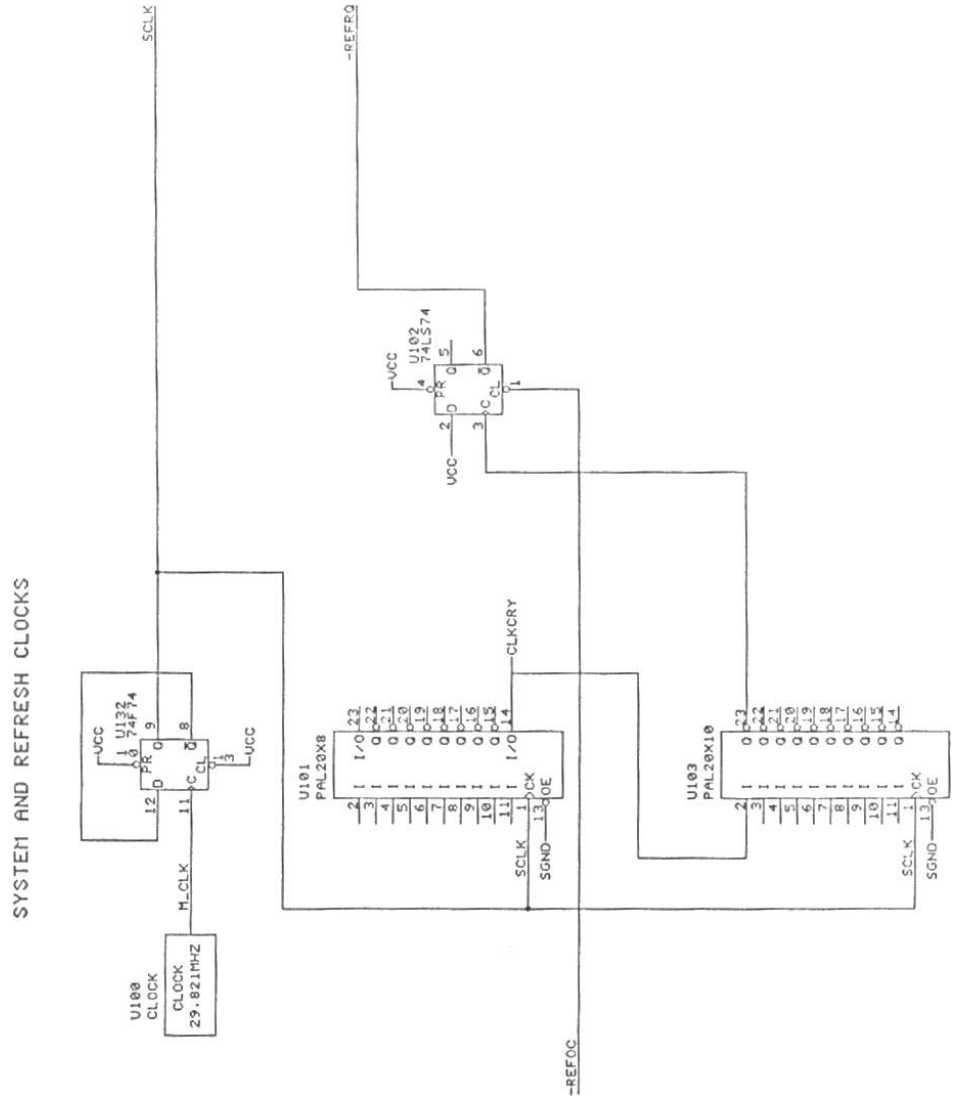

Figure B.10: System and Refresh Clocks

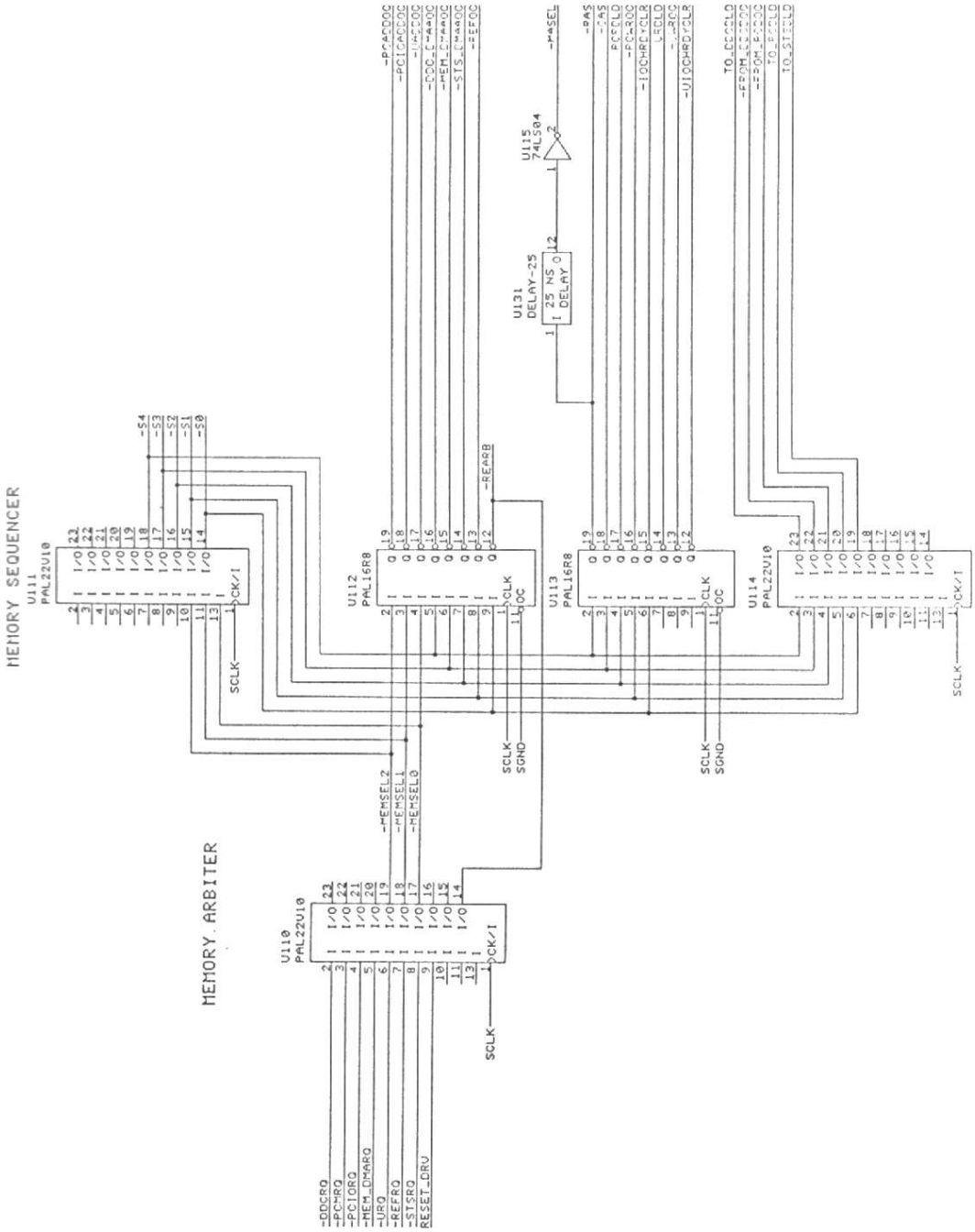

Figure B.11: Memory Arbiter and Sequencer

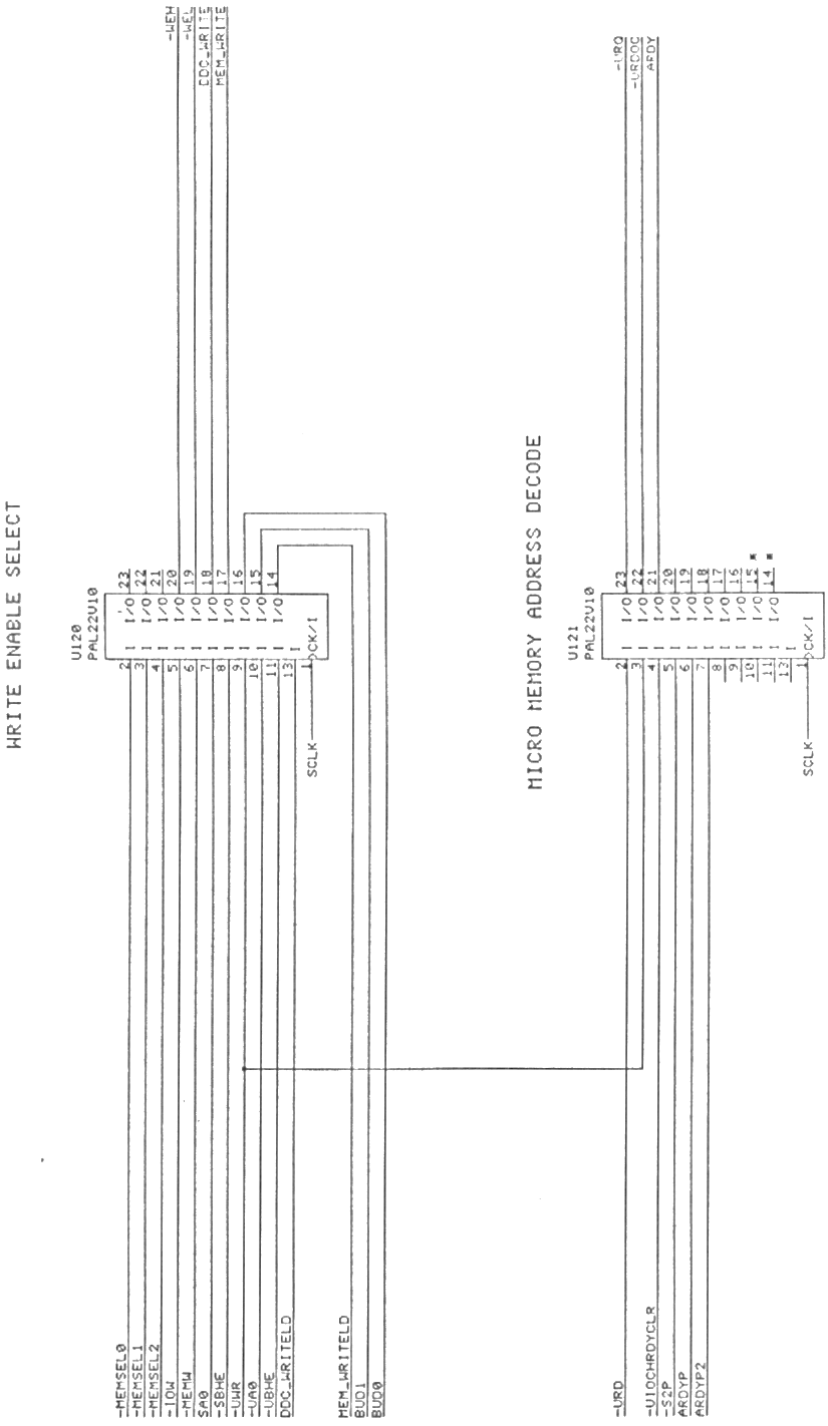

Figure B.12: Write Enable Select and Micro Memory Address Decode

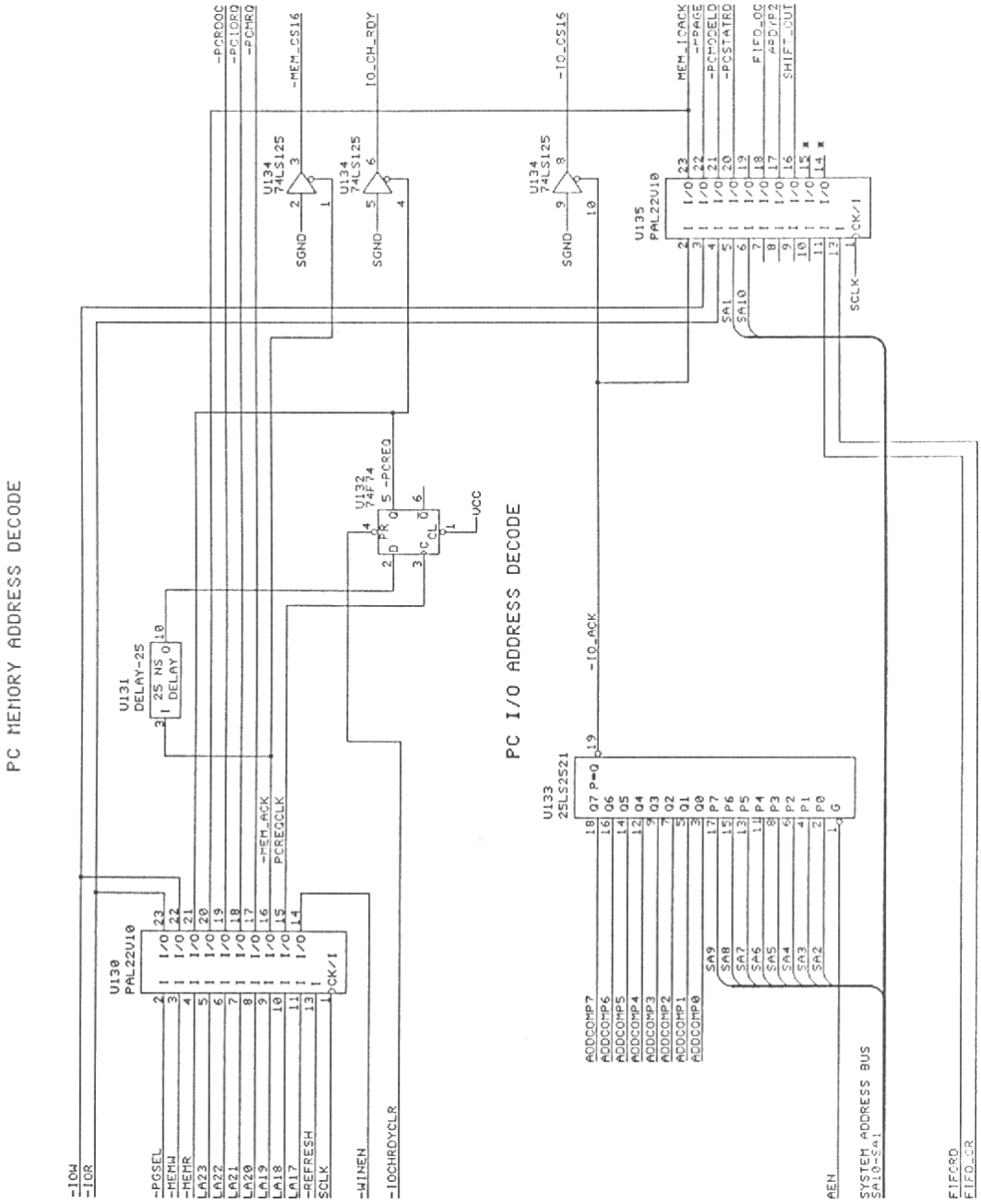

Figure B.13: PC Memory and I/O Address Decode

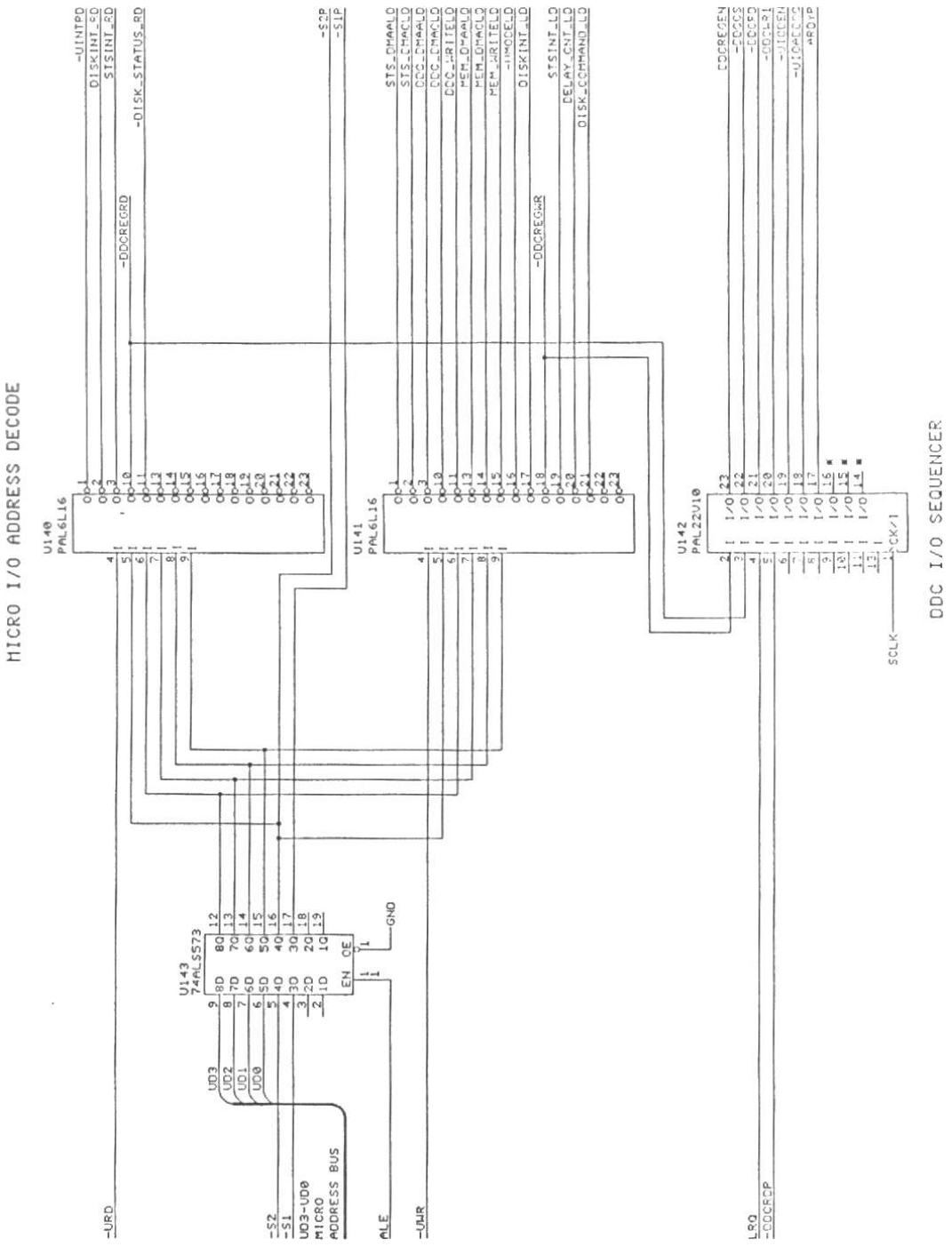

Figure B.14: Micro I/O Address Decode and DDC I/O Sequencer

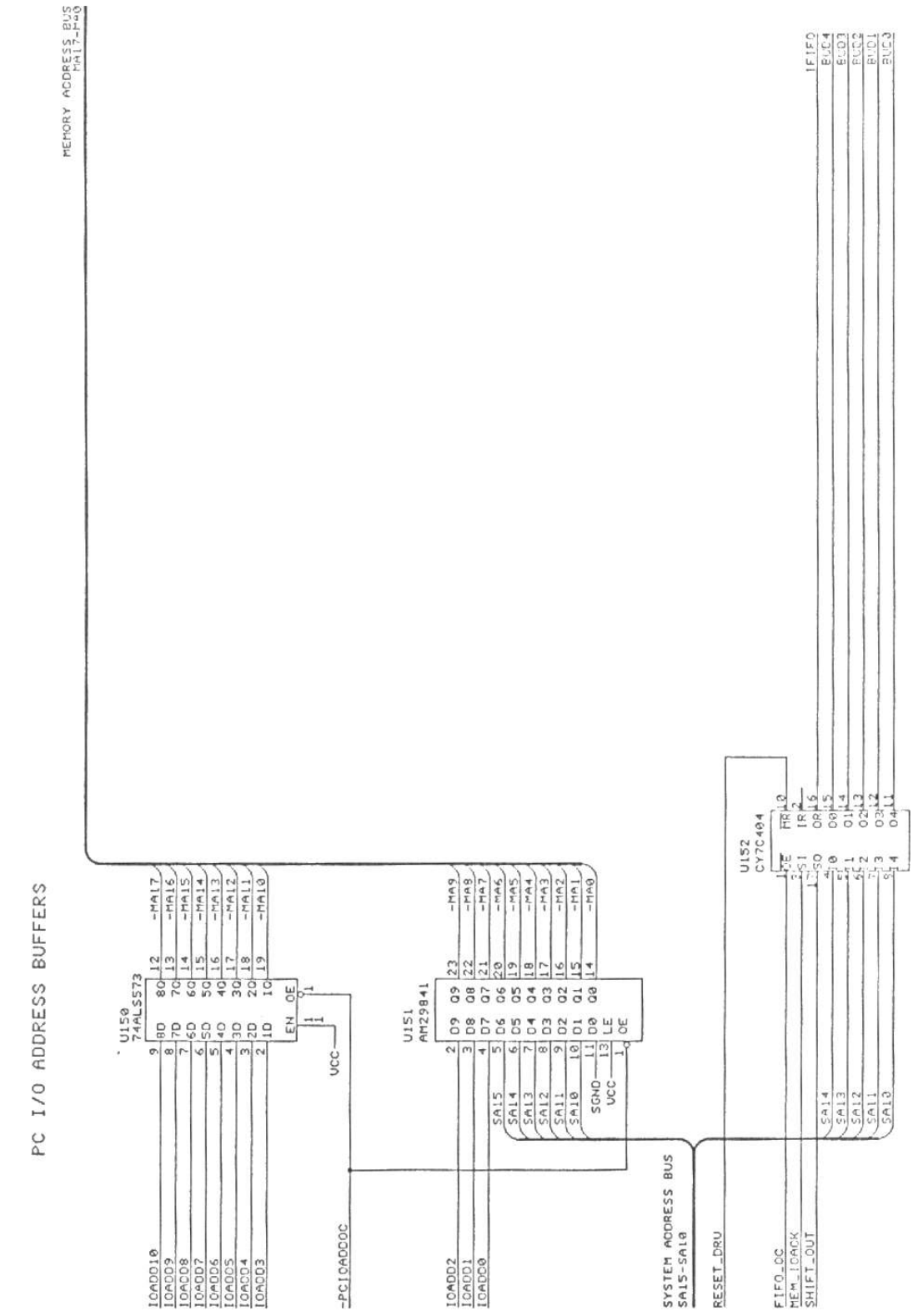

Figure B.15: PC I/O FIFO and Buffers

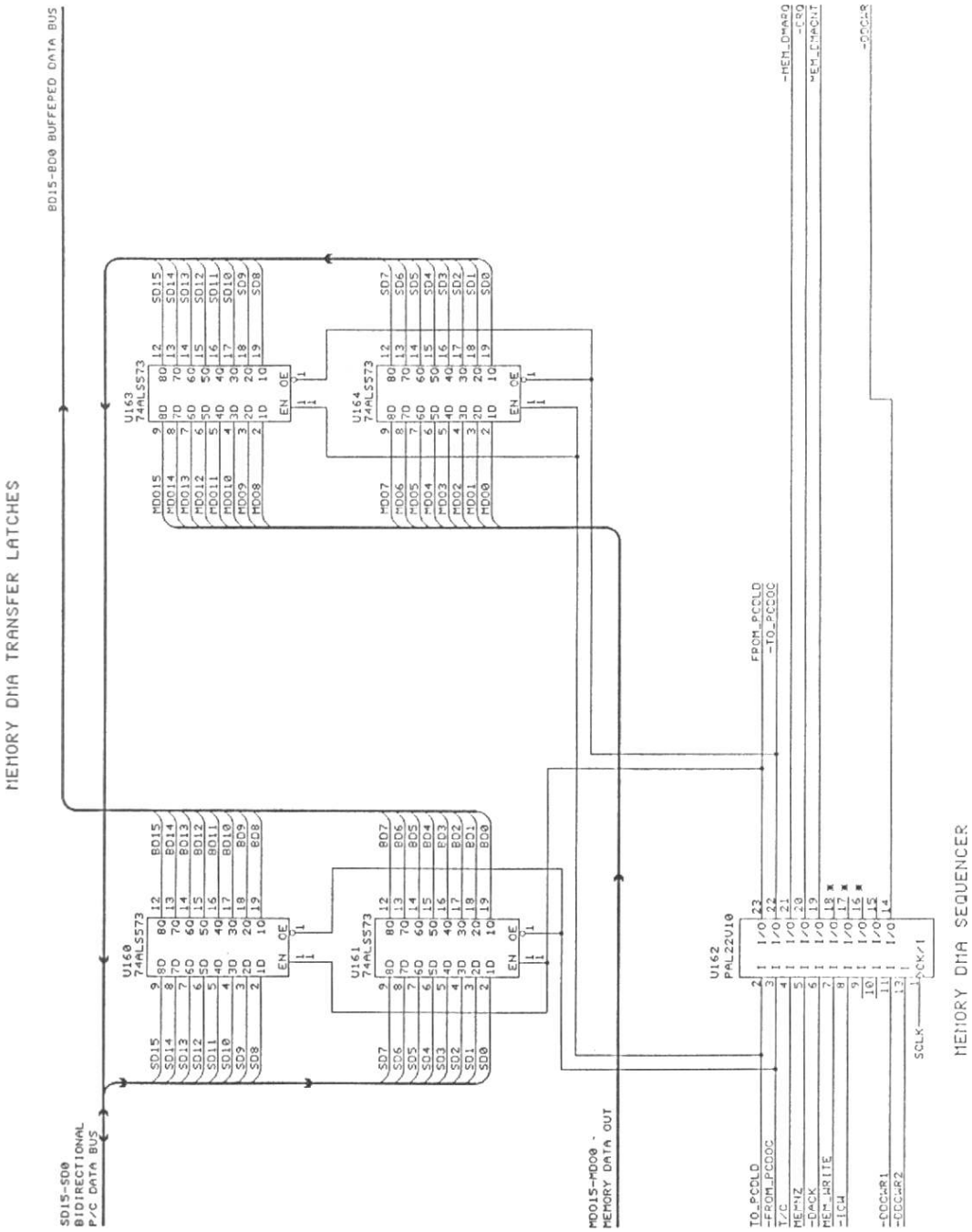

Figure B.16: PC DMA Sequencer and Transfer Latches

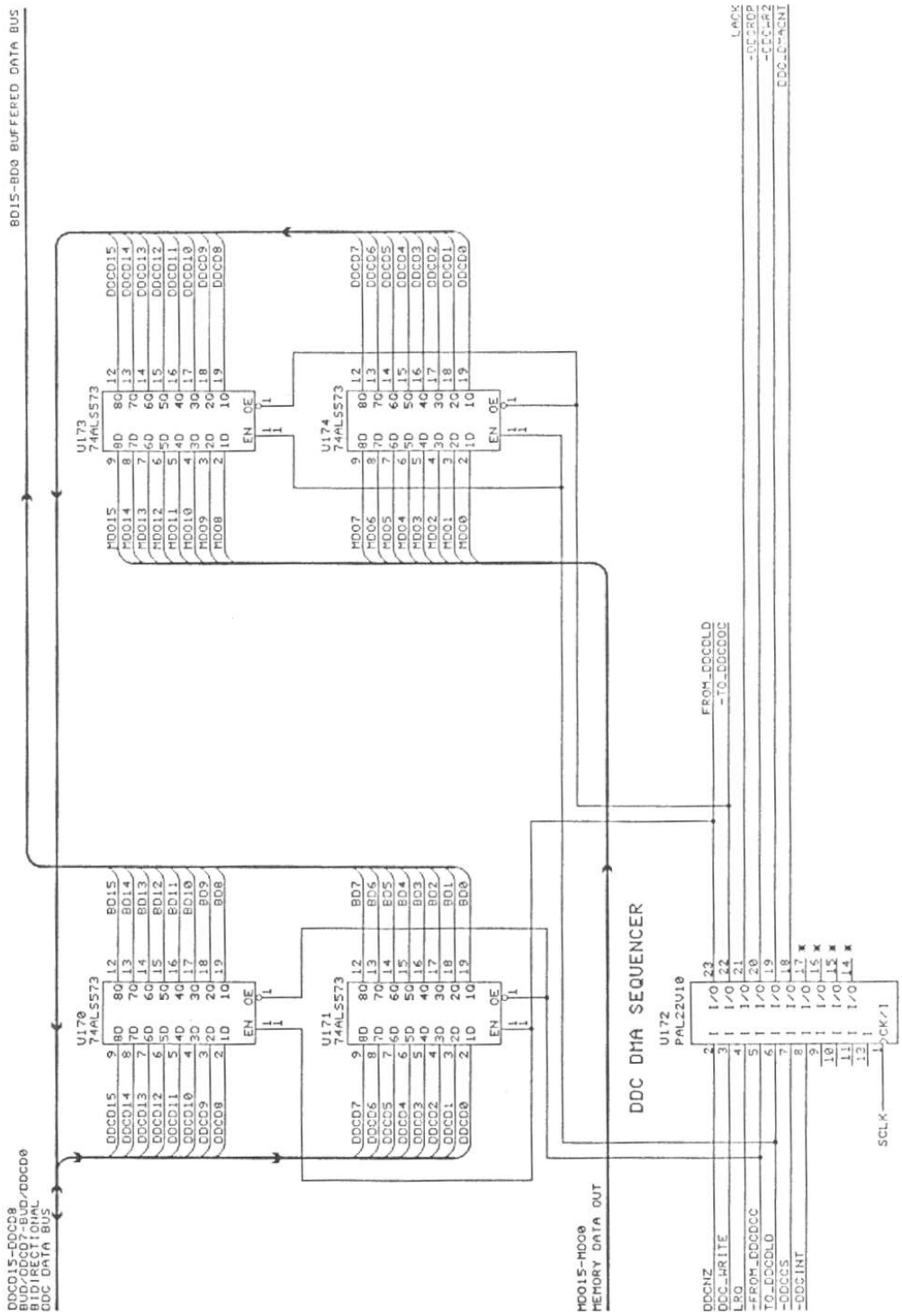

Figure B.17: DDC DMA Sequencer and Transfer Latches

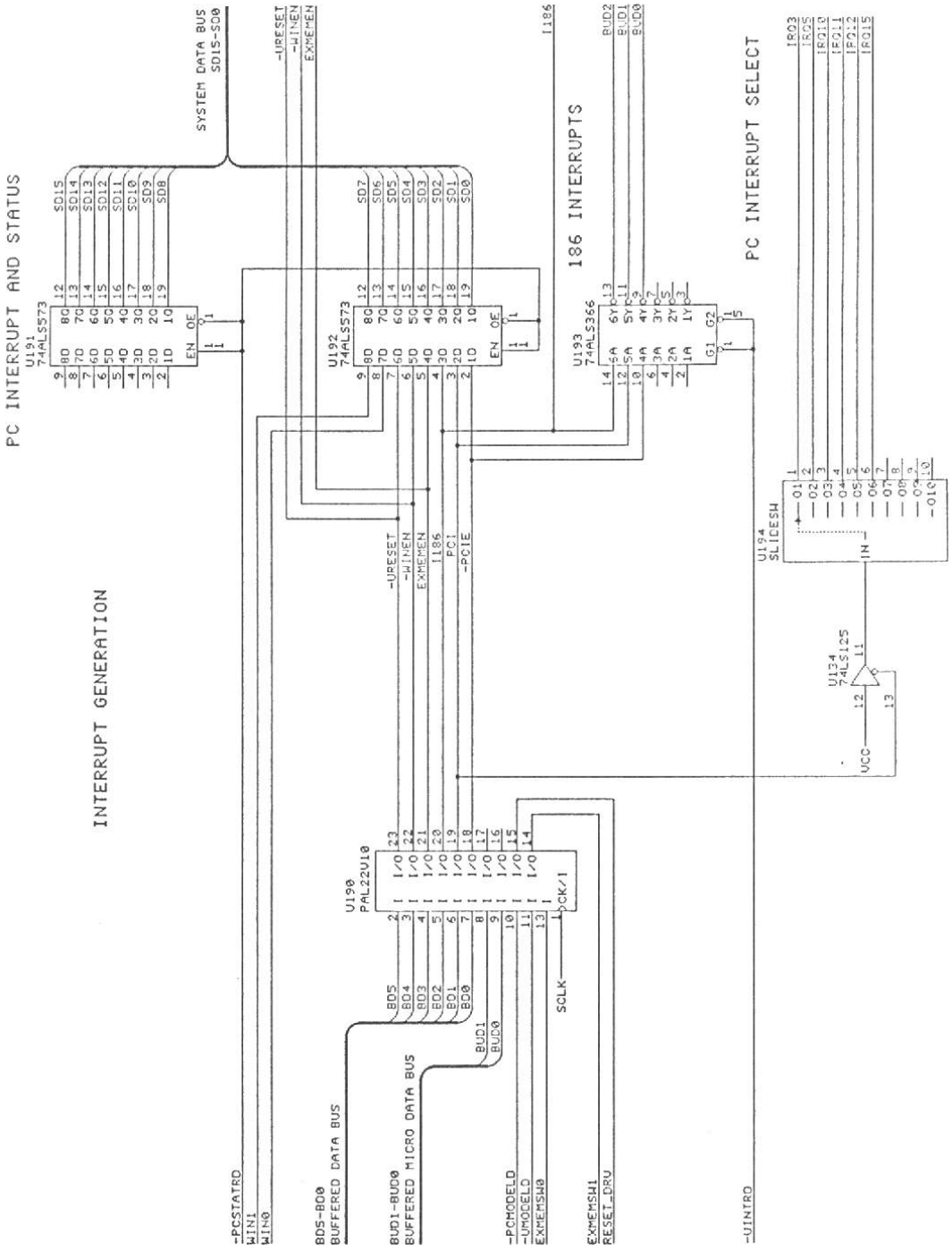

Figure B.18: Interrupt Generation

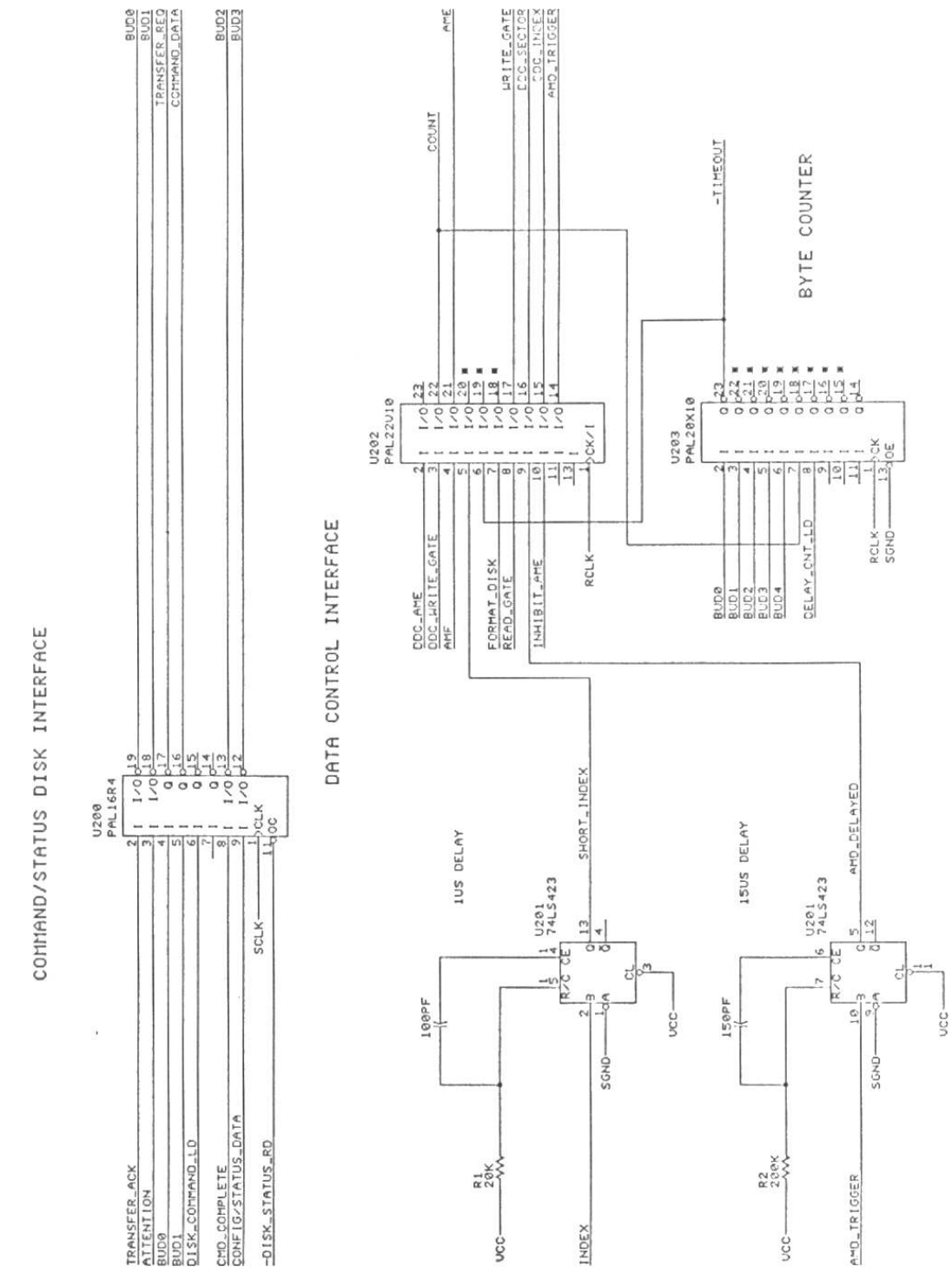

Figure B.19: DDC-Disk Interface

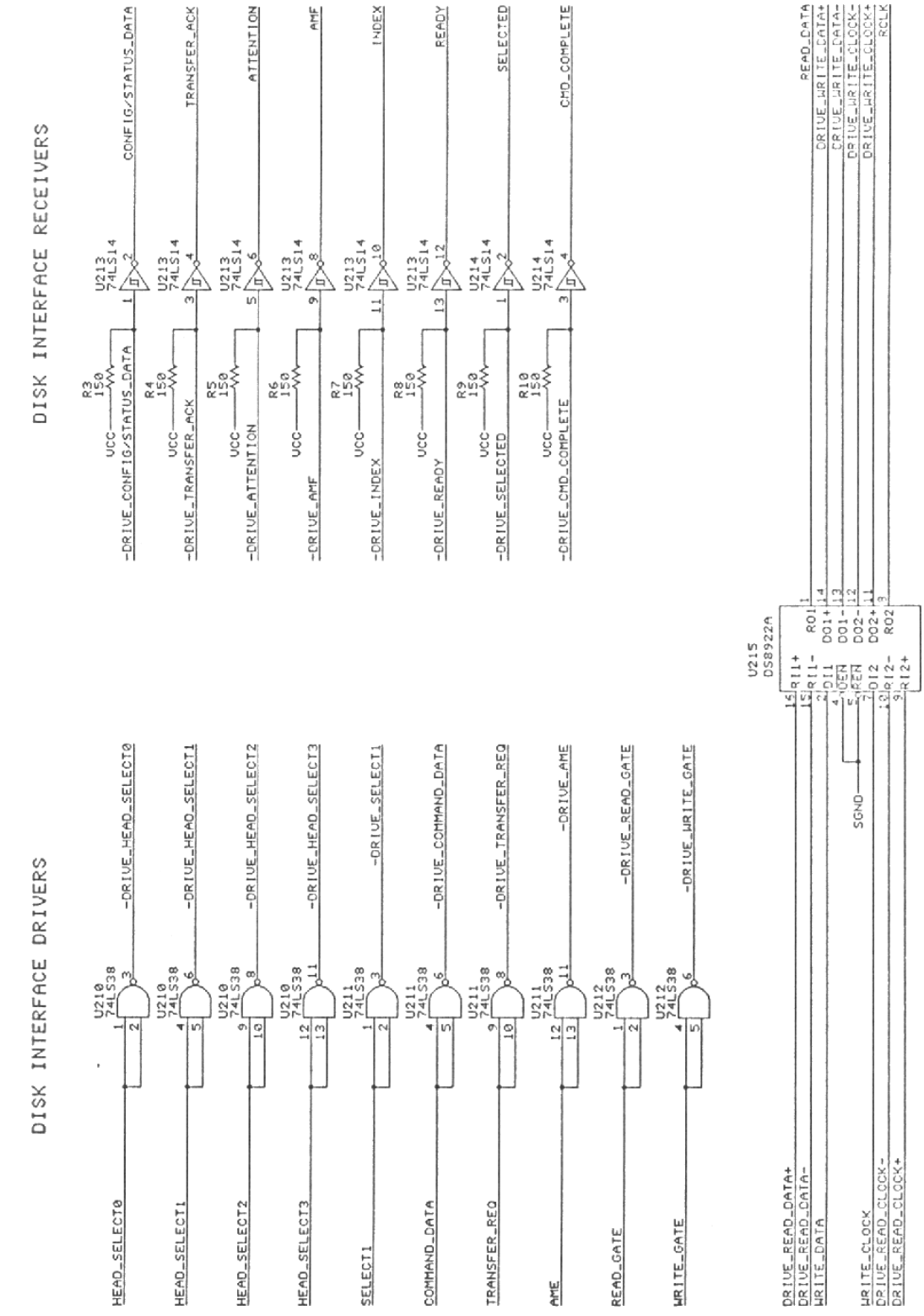

Figure B.20: Disk Interface Drivers and Receivers

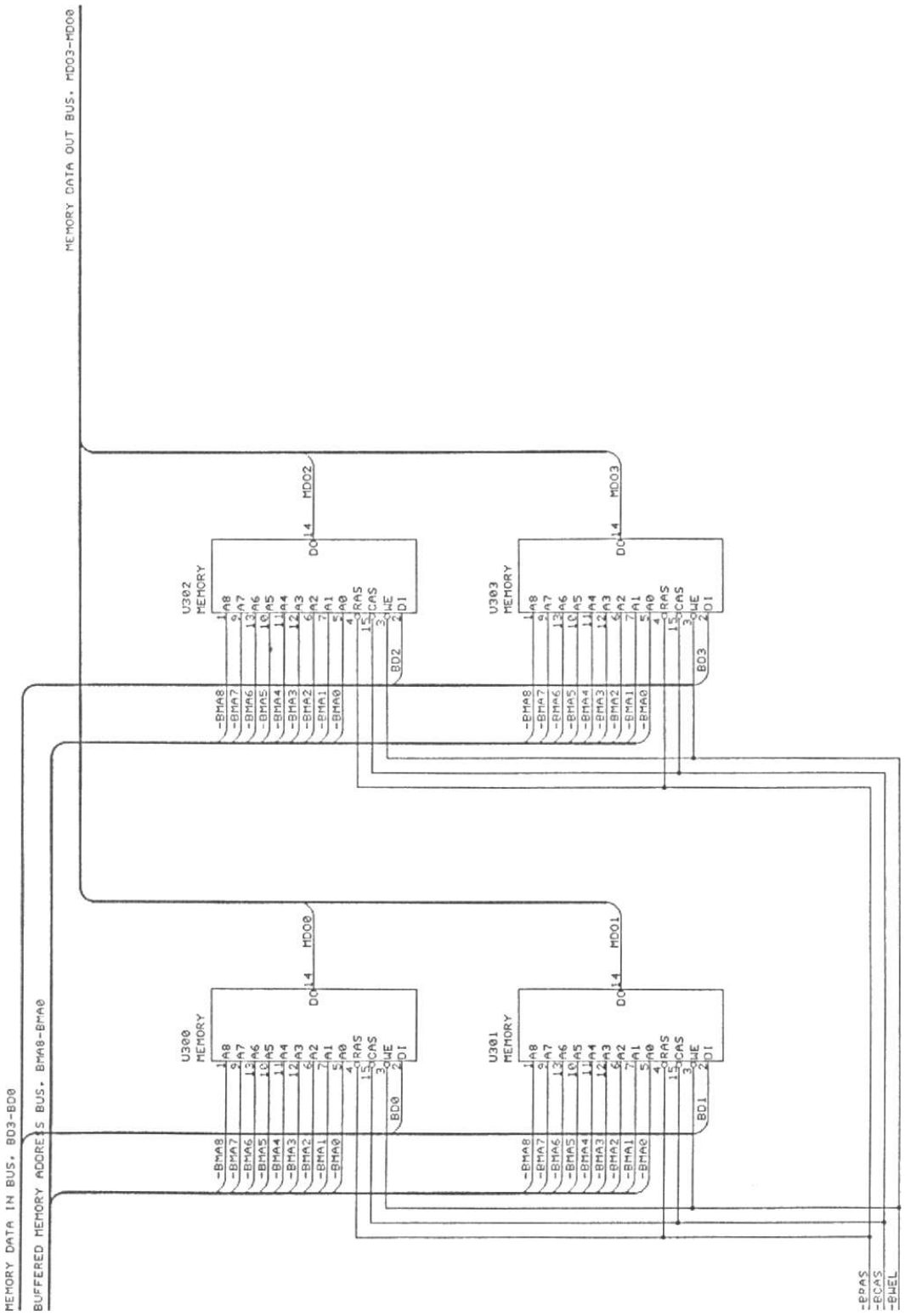

Figure B.21: Memory-First Four Bits

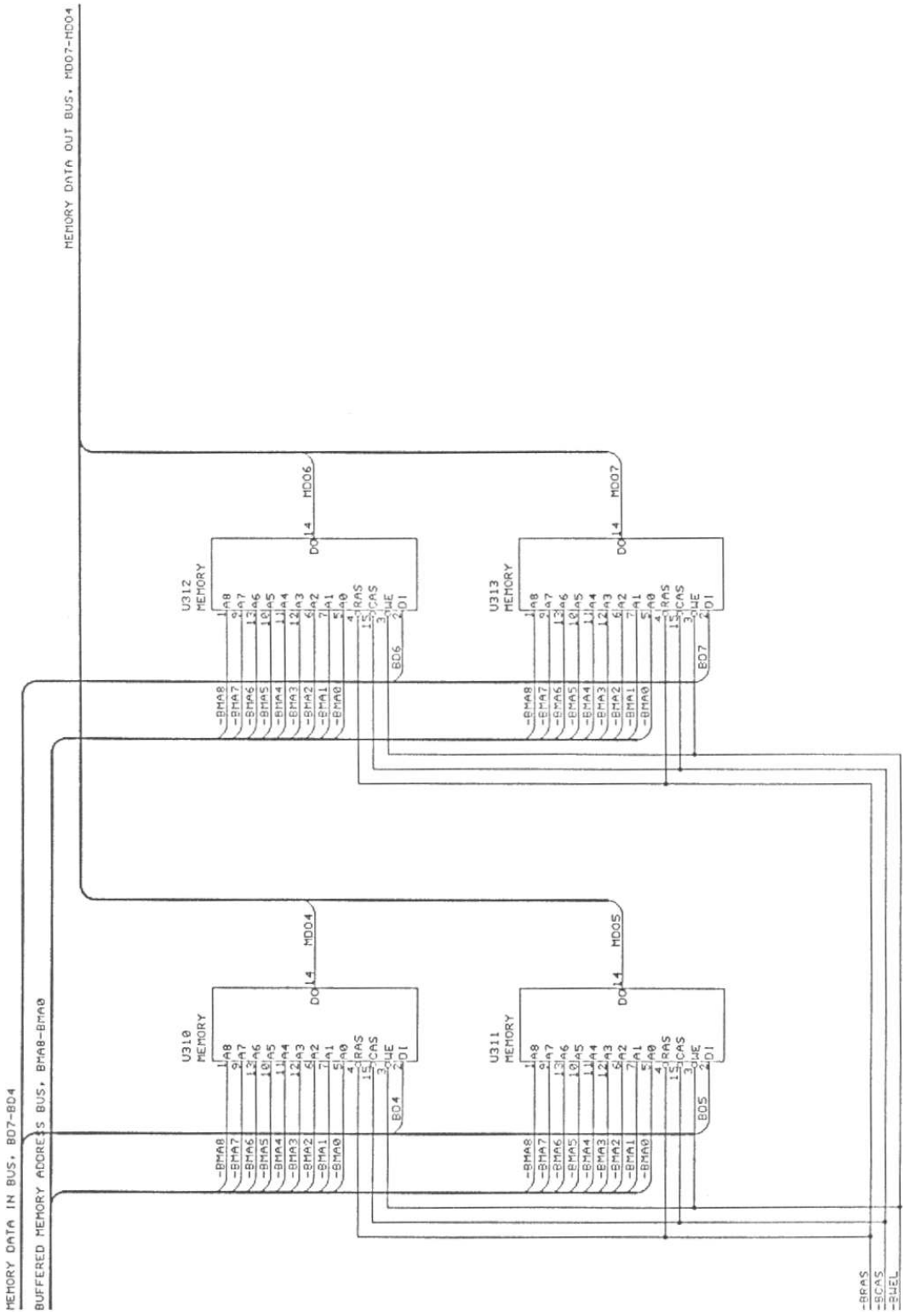

Figure B.22: Memory-Second Four Bits

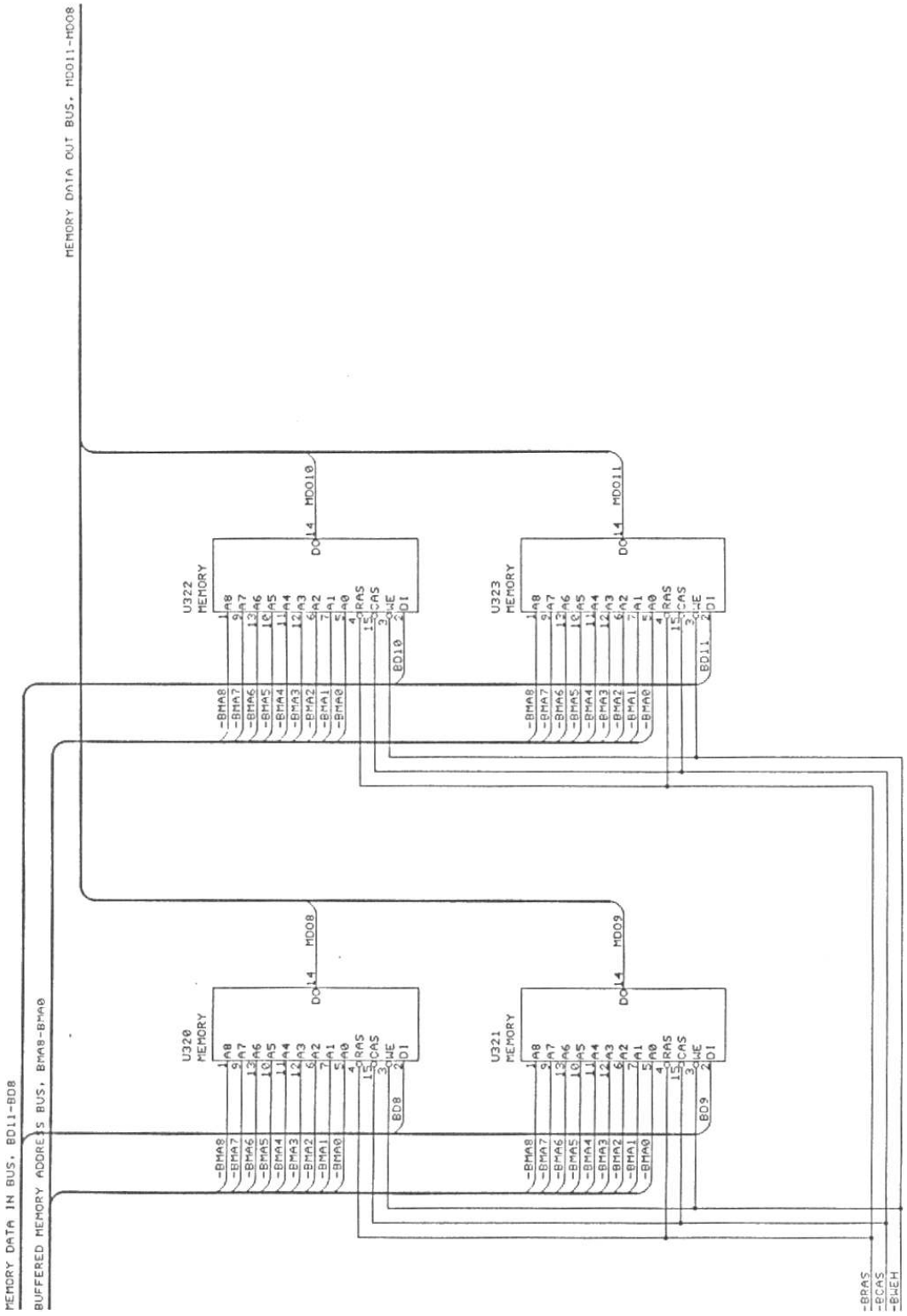

Figure B.23: Memory-Third Four Bits

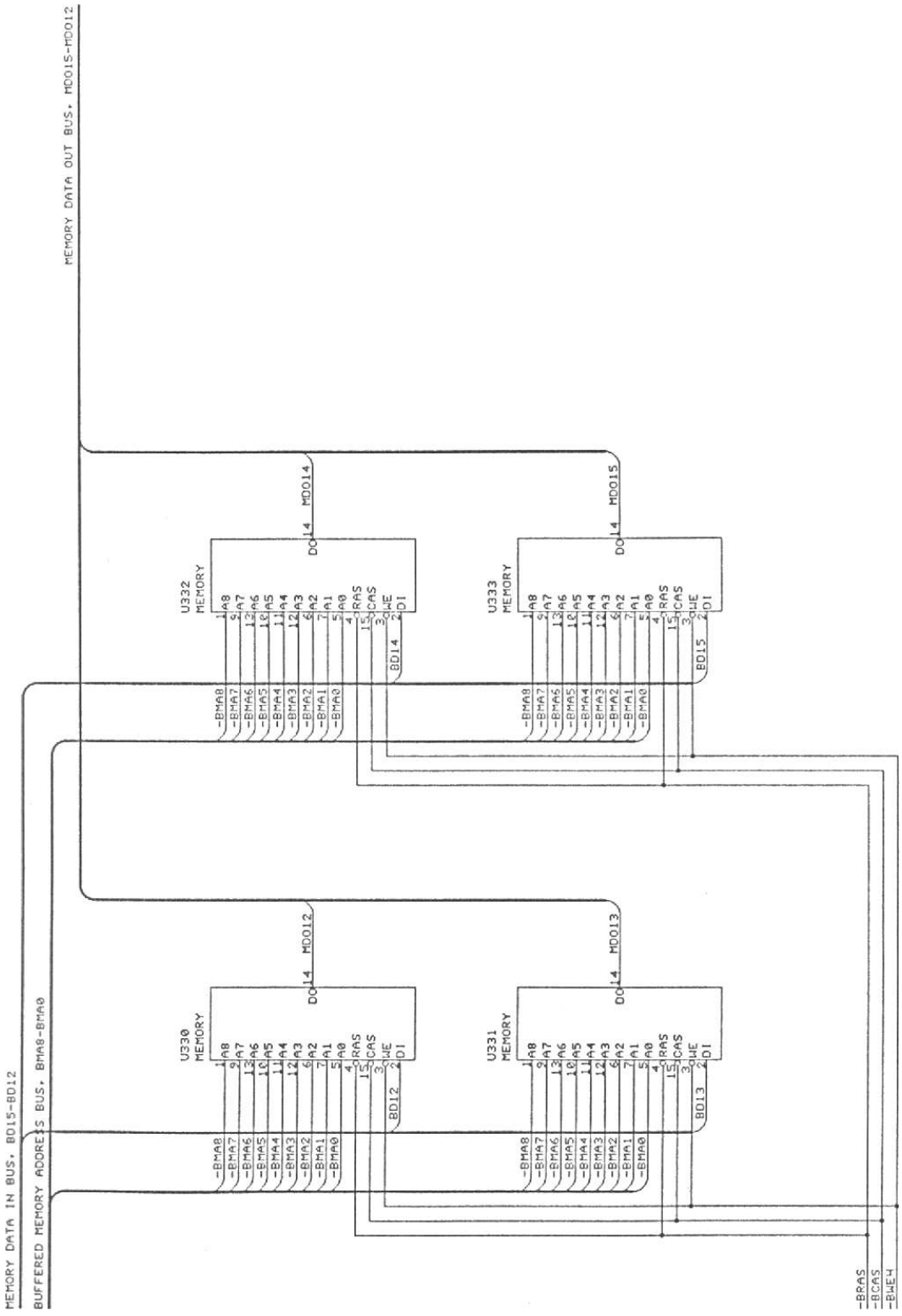

Figure B.24: Memory-Fourth Four Bits

APPENDIX C

SYSTEM-CHIP INTERFACE CIRCUIT SCHEMATICS

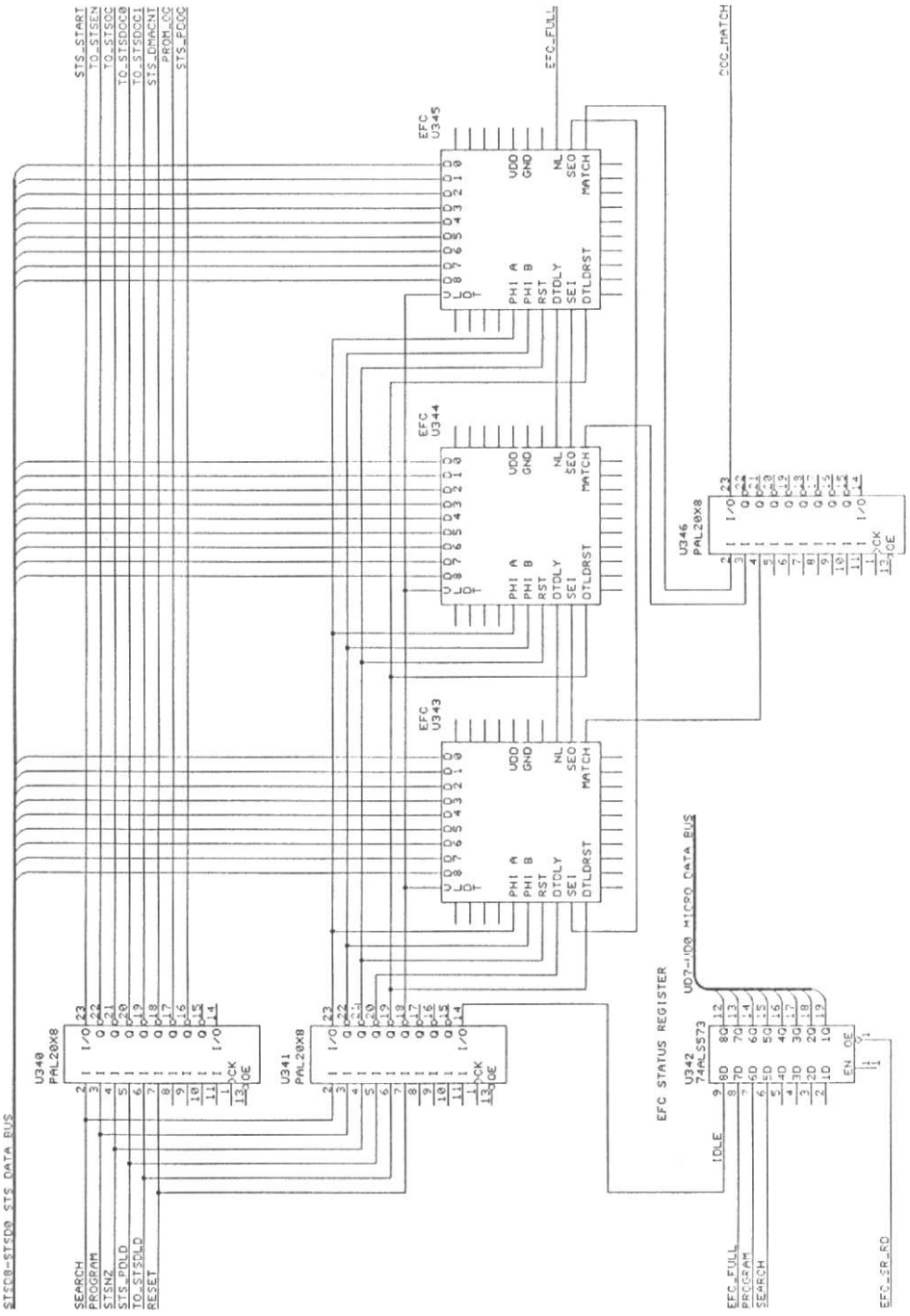

Figure C.1: EFC Sequencer and EFC Chips

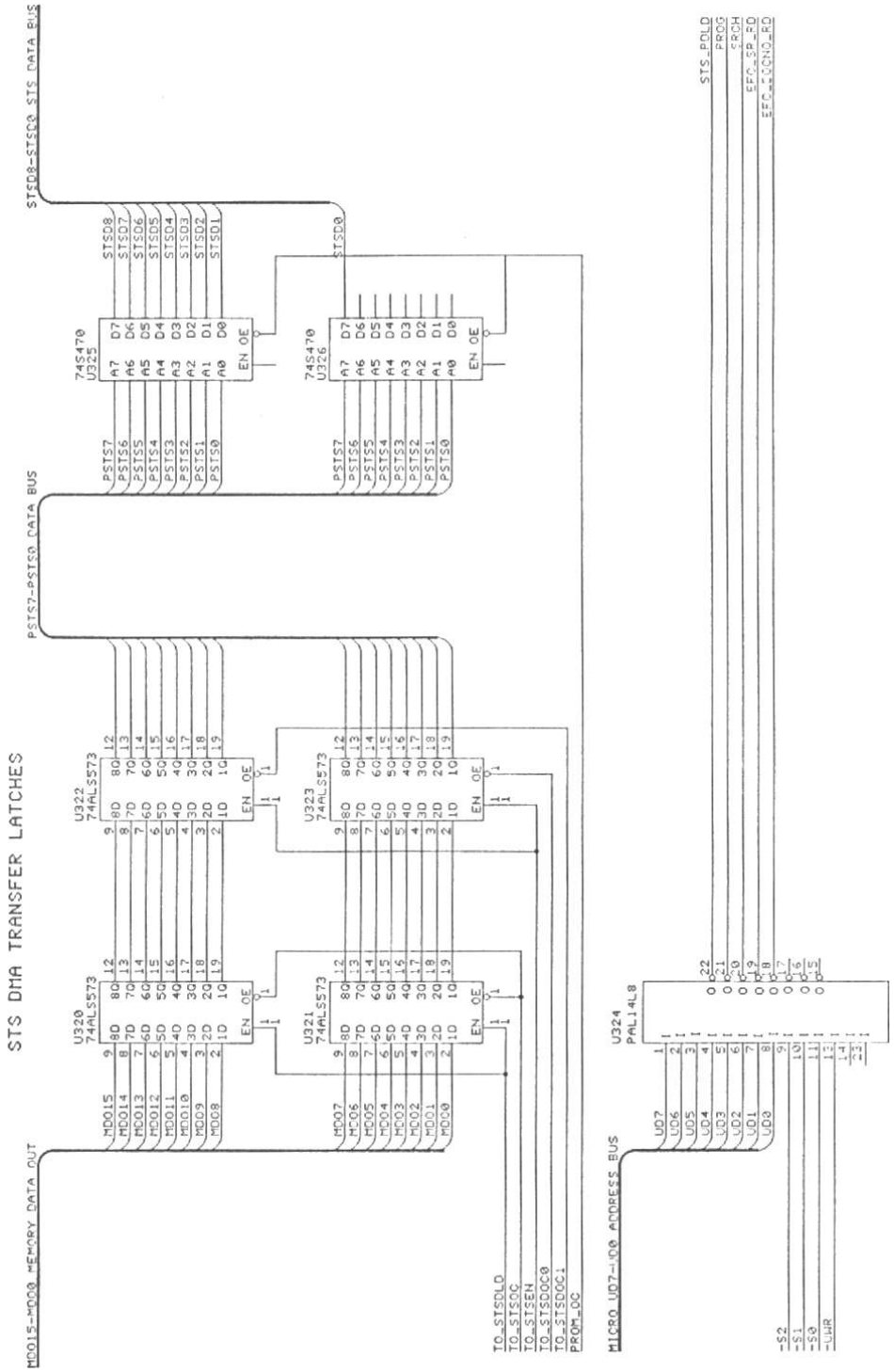

Figure C.2: EFC DMA Transfer Latches and PROM

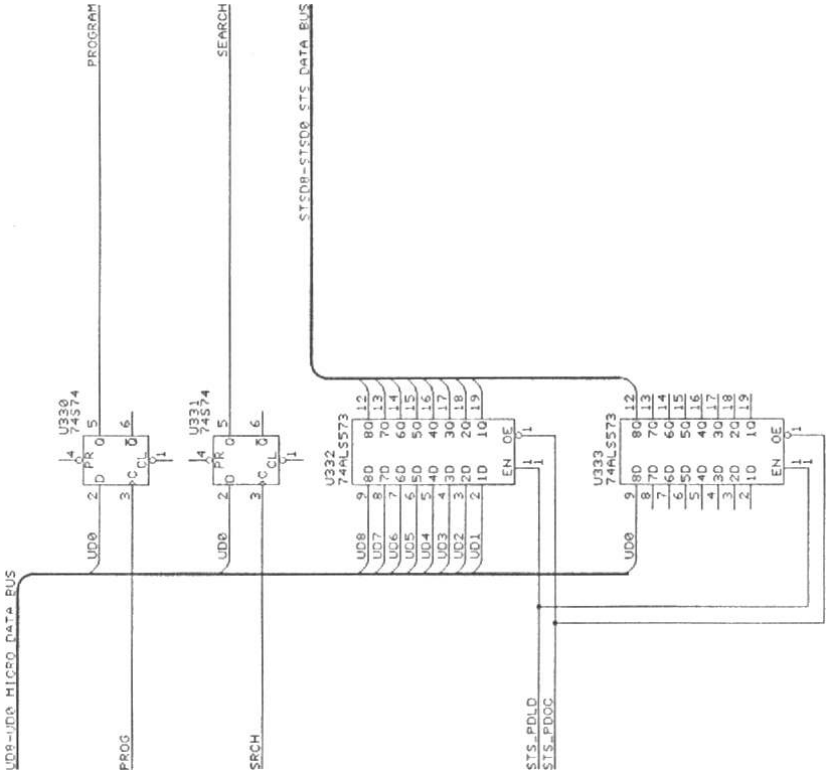

Figure C.3: EFC 186 Transfer Latches

BIBLIOGRAPHY

[1] *Microprocessor and Peripherals Handbook*. Intel Corporation, Intel Corporation Literature Department, 3065 Bowers Avenue, Sabta Clara, CA 95051, 1983.

[2] Andrew R. Jacob. *Hard Disk Controller for an Electronic File Cabinet*. University of Delaware, 1988. Undergraduate Research.

[3] Donald W. Mules. *An Electronic Filing Cabinet*. University of Delaware, August 1983. Master's thesis.

[4] Jeong-Gyun Shin. *A CMOS VLSI Implementation of an Electronic Filing Cabinet*. University of Delaware, December 1988. Master's thesis.

[5] Peter J. Warter. *A Proposal for an Electronic Filing Cabinet*. Technical Report, IEEE Computer Society, Micro Delcon, March 1979.