1988

A coprocessor design for the architectural support of non-numeric operations

Timothy W. Curry
University of Central Florida

A COPROCESSOR DESIGN FOR THE ARCHITECTURAL SUPPORT OF NON-NUMERIC OPERATIONS

by

TIMOTHY W. CURRY

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
the Department of Computer Science at
the University of Central Florida
Orlando, Florida

August 1988

Major Professor: Amar Mukherjee

ABSTRACT

Computer Science is concerned with the electronic manipulation of information. Continually increasing amounts of computer time are being expended on information that is not numeric. This is represented in part by modern computing requirements such as the block moves associated with context switching and virtual memory management, peripheral device communication, compilers, editors, word processors, databases, and text retrieval. This dissertation examines the traditional support of non-numeric information from a software, firmware, and hardware perspective and presents a coprocessor design to improve the performance of a set of non-numeric operations.

Simple micro-coding of operations can provide a degree of performance improvement through parallel execution of instructions and control store access speeds. New special purpose parallel hardware algorithms can yield complexity improvements. This dissertation presents a parallel hardware regular expression searching algorithm which requires linear time and quadratic space compared to software uniprocessor algorithms which require exponential time and space. A very large scale integration (VLSI) implementation of a version of this algorithm was designed, fabricated, and tested. The hardware searching algorithm is then combined with other special purpose hardware to implement a set of operations. Simulation is then used to quantify the performance improvement of the operations when compared to software solutions.

A coprocessor approach allows the *optional* addition of hardware to accelerate a set of operations. This is appropriate from a complex instruction set computer (CISC) perspective since hardware acceleration is being utilized. It is also appropriate from a reduced instruction set computer (RISC) perspective since the operations are distributed away from the central processing unit (CPU).

ACKNOWLEDGEMENTS

The patience and support of friends, family, coworkers, and my committee are the key factors that helped me through this dissertation. I would like to thank Dr. Amar Mukherjee for his solid and long term support of this work. His insight, commitment to quality, and personal efforts have been invaluable in shaping and developing this dissertation. Likewise, the changes, suggestions, and input from Dr. Charles Hughes, Dr. Ratan Guha, Dr. Brian Petrasko, and Susanne Payne have greatly enhanced and improved the contents of this dissertation.

I would also like to thank Dave Phillips and Clay Johnson for continually intriguing discussions which inspired some of the directions this work took. Without question, the personal and academic comradery of Dr. Alan Eustace has kept me going, and his friendship continues to be closely valued. Personal thanks are also given to Jim Davis, Paul Hale, and Mike Geldner. Jim's commitment to the continued educational and personal development of his staff contributed more than he knows towards my attaining this goal. Mike's personal and professional assistance was incredible and he has my greatest gratitude.

The typographical, editorial, and bibliographical assistance of my wife, Amy, is reflected in this dissertation, but the never failing support and companionship she has provided is reflected in my life. Additionally, I would like to thank Vikki Irwin and my father, Ken Curry, for their unwavering support in all aspects and directions I have taken.

This work was supported in part by NSF grant MCS-8005096A02.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

In the broadest concept of computing, all computer applications can be described as the electronic manipulation of information. From the simplest clocks and calculators to the most advanced super computer simulations and process control applications, the bottom line is that information is used as input to an application and manipulated in some fashion. Computers and their applications are differentiated by the types of information they are manipulating and how that manipulation is accomplished. An initial dichotomy of types of information can be divided into numeric and non-numeric, that is to say, numbers and everything else.

The computer industry is currently dividing central processing unit (CPU) designs into another dichotomy. One camp advocates the support of sophisticated data manipulations in the CPU hardware design and the other camp advocates a simple CPU design with most data manipulation performed in software. Both camps will agree that hardware acceleration can yield significant improvement in the execution of operations but they disagree where, when, and how that hardware acceleration should be accomplished. The use of a secondary processor called a *coprocessor* to augment the CPU functionality has become a common solution. Numerically oriented floating point coprocessors are very common in the marketplace and are utilized by both CPU design camps. This dissertation examines and demonstrates the viability of a coprocessor design to support a set of non-numeric operations.

1.1. Non-Numeric Data

Numeric information manipulation historically drove the designs of the first electronic computers (Kuck 1978), and the performances of various numeric operations are still the primary factors in determining the computing power of modern computer systems (Linton 1986). However, as computer technology has advanced, its uses, demands, and applications have spread out into a variety of areas.

The use of computers for text retrieval and large databases represents the most glaringly obvious requirements for large scale use of non-numeric operations. These applications are becoming more common as disk storage technology improves, especially with the high densities and relatively low cost associated with optical medium. The American Institute of Industrial Management (AIIM) estimates that a quarter of a billion pages of original documents are created every day (Leerburger 1988) and that number jumps to 3.4 billion pages per day when copies and computer printouts are incorporated into the calculation. AIIM further estimates that corporate documents are increasing at 20% per year. While not all of this information is generated or stored electronically, continually increasing portions are available electronically.

Some of the well known retrieval systems (Bayer 1978; Black 1978; Larson 1977; McCarn 1978; Sprowl 1976) currently provide electronic access to professional journals and databases such as the medical profession's MEDLINE system and the legal profession's LEXIS system, each representing gigabytes of information. One of the most ambitious examples in this genre is being implemented by the U.S. Department of Commerce, Patent and Trademark Office. It is currently in the process of electronically storing 40 million pages of domestic patents and over 60 million pages of foreign patents (Leerburger 1988). This represents more than a terabyte of information which will be electronically manipulated.

When manipulating just kilobytes of information, a simple doubling in the performance of an operation can yield a significant improvement. A complexity improvement in the basic algorithm of an operation would yield tremendous results at these levels of data processing. The megabyte, gigabyte, and terabyte examples presented here are the extreme ends of non-numeric requirements but do show that such requirements exist and are real. Further justification for hardware support of non-numeric operations can be established by examining how deeply rooted and unavoidable non-numeric operations have become in the general computing environment.

The history of computing shows an on-going process to make the computer a better tool by making it easier for humans to use. This process has invariably been accomplished by making the computer do more work. No one would believe that computers would be used in as many varied applications as they are today if the computer user still had to compute the binary for each machine instruction and use toggle switches to enter one instruction at a time. Instead, through various stages, we have reached a point where bit mapped displays and pointing devices, like a mouse, are becoming mandatory for interactive work with a computer. It even appears that speech recognition may be a realizable goal in the near future. The trend is to allow the human to communicate with the computer in the most comfortable and natural fashion possible. Advances in memory and storage devices have allowed vast amounts of information to remain readily available to the computer while assemblers, compilers, and other software tools have been developed to transform human convenient abstractions to the necessary binary format that the computer requires. These human readable formats also tend to be more portable than the machine dependent binary formats and therefore are often used as a means for transferring information between systems or programs. The computers of the past defined the environment in which the computer user worked. The computers of today must compete in a market that requires speed, ease of use, availability, reliability, and maintainability for both the hardware and software.

A modern computer must be concerned with multi-tasking, virtual memory management, network and peripheral communications, and a vast variety of other processes in addition to an individual program being run. Each of these duties tends to require movement of information without concern for what the information is. Multi-tasking requires context switching, virtual memory requires paging, network and peripheral communications require transfers of blocks of information. The assemblers, compilers, and block movement requirements described so far gain very little from how fast two numbers can be added or multiplied. Neither do applications such as interactive command processors, editors, word processors, spelling checkers, electronic mail, text retrieval, databases, or a number of other textually oriented applications which are becoming an ever increasing portion of the utilization of electronic computing.

The distinguishing attribute of numeric operations when compared to non-numeric operations centers on the size of the data. The numeric operations which are commonly implemented in hardware are generally unary or binary, and all of the operands are able to fit in fast access memory locations (registers). Non-numeric operations are concerned with large and often unknown amounts of data which can not be held in a register. For example, something as simple as scrolling the information on a bit mapped display can incur considerable overhead on a system. Consider a 1024 x 1024 pixel color screen (one megapixels) with 8-bits per pixel which needs to shift up by one raster line. This implies the movement of a megabyte of data. Given a machine which could execute a million instructions per second, has an instruction that could do a memory to memory byte transfer, and another instruction that could decrement a counter and conditionally branch, this operation would take just over two seconds to accomplish, provided the entire machine was dedicated to doing only the scroll. Two seconds is intolerably long to accomplish the scroll in the first place, and consuming the entire system for that long for such a simple task would be unacceptable as well. The solution to this is to add special purpose hardware that can perform the scroll operation independently of the demands of the central processing unit (CPU) and perform it at a more acceptable rate of at least four times faster.

1.2. Software versus Firmware versus Hardware

The solution of special purpose hardware for a particular task is quite common in the computer industry. Disk and tape drives come with controllers; graphical displays have dedicated memory and dedicated hardware to manipulate that memory; virtual memory systems have hardware memory management units (MMUs); array processors are available to accelerate applications which work on large arrays of numbers; and multitudes of floating point accelerators exist.

There is nothing accomplished by these special purpose hardware designs that could not also be done by the CPU, but as demonstrated by the scrolling example of the previous section, the speed at which it can be accomplished is an important factor. The purchaser of a modern computer must determine how critical the support and speed of a particular data operation is to the application. Extending this example to the extreme, there is nothing a multi-million dollar super computer can compute that an inexpensive home personal computer could not also compute. The difference is that the super computer computes it **much** faster. There is a point of diminished return between cost and speed associated with the movement of an application from a slower machine to a faster machine or from the control of the CPU to the control of special hardware.

The primary problem with non-numeric operations is the limitation of processing small amounts of the information at a time in a loop. A speed penalty is paid since both the program performing the operation and the data on which the operation is being performed are in memory. There is not much that can be done about the data being in memory, but the program can be moved into hardware.

If the program can make use of the existing hardware with no additions or changes, then a *firmware* solution is possible. Some machines like the DEC VAX 11/780 (1977) have a loadable control store available to their users to add custom machine instructions. New CPU designs may also use this technique during the definition and debugging of their machine. Some of the operations which will be designed in this dissertation could be accomplished through simple microprogramming of a general purpose computer architecture. Other operations will require special hardware that is not commonly found in CPU designs. This dissertation will attempt to provide insight into the gains that can be achieved by movement from a pure software implementation to a firmware implementation, and the complexity of any special hardware requirements.

1.3. CISC versus RISC versus Coprocessors

Moving more and more operations into the firmware of the CPU has been the trend in the computer industry for several decades. The increased functionality offered by Very Large Scale Integration (VLSI) technology presents a new opportunity to build special purpose hardware for reliable and high-performance machines. There are a variety of different approaches being taken by researchers and manufacturers attempting to exploit VLSI technology. One approach is to build increasingly complex microprocessors by taking advantage of the increased level of integration to implement more complex but reasonably established architectures (Intel 80386, Motorola MC68030, National Semiconductor NS32032, Western Electric WE32000). These are called complex instruction set computers (CISC).

A different approach advocates precisely the opposite of complex processors. Some researchers maintain that the speed gap between memory and the processor is no longer significant and that compilers are unable to effectively utilize the "exotic instructions" made available by the complex processors. Instead, they advocate a simplified CPU design realizing a speed improvement by utilizing the chip area for pipelining, caching, etc. This design philosophy, referred to as reduced instruction set computers (RISC), has been adopted in the Berkeley RISC processor (Fitzpatrick 1981), the IBM 801 processor (Radin 1982), and the Stanford MIPS processor (Hennessy et al. 1982) and has found its way into commercial processor designs such as those offered by Pyramid, Sun (1987), MIPS (Moussouris et al. 1986), and HP (Birnbaum and Worley 1985).

Another approach seeks to exploit parallelism in the hardware to perform functions that are traditionally performed inefficiently on a uniprocessor. Research in non-vonNeumann architectures is quite extensive, but a good subset of algorithms investigated for VLSI as well as a list of references can be found in Chapter 8 of the Mead-Conway text (1980).

This dissertation will present the design of a coprocessor which manipulates character string data. Utilizing a coprocessor approach to optionally provide CPU enhancements avoids increasing the instruction decode time of the CPU (a reduced instruction set versus complex instruction set consideration) but provides high speed hardware support for complex operations (a complex instruction set versus reduced instruction set consideration). Additionally, one of the string operations presents a traditionally exponential time and space problem that is shown to have a practical quadratic space and linear time realization utilizing a parallel processing technique in VLSI.

1.4. Structure of the Dissertation

Chapter 2 presents the set of non-numeric operations which will be implemented, providing details on the structure of the data and traditional use and implementation of these operations. One of the most frequent and complex operations is searching. Chapter 3 is dedicated to this one operation presenting a powerful and practical special purpose hardware algorithm for implementing a searching algorithm. Chapter 4 incorporates the searching algorithm with other hardware to implement the full set of operations. Chapter 5 simulates the coprocessor design to quantify the gains of software to firmware to hardware. The results of this simulation and the coprocessor concept are then summarized in the conclusions of Chapter 6.

CHAPTER 2

THE SELECTION OF NON-NUMERIC OPERATIONS

Attempting to discern the requirements of **all** non-numeric operations and designing hardware implementations would be impractical. Furthermore, some non-numeric operations such as the device controllers, intelligent graphic displays, and memory management units (MMUs) mentioned in the previous chapter are already realized in hardware. Instead, we submit that character string operations are representative of the majority of non-numeric work performed by computers. Thus, examination of hardware acceleration for character string operations provides evidence of the types of gains that could be expected for many other non-numeric operations. All non-numeric operations are implemented with loops which operate on large amounts of data. In the case of character strings, the assumption is made that the data consists of characters, but the operations are often applicable to other non-numeric operations as well. For example, the page movement operation of an MMU and the scrolling operation of a graphic display both accomplish the same operation as the *strcpy* (string copy) operation of character strings. In fact, the MMU has an easier job since it is moving a fixed, known amount of data. By quantifying the improvements of string operations moved into hardware, a basis is established for the improvements that might be expected for similar operations.

This chapter selects a set of character string operations, reviews existing string languages, reviews traditional hardware support of strings, and discusses how the selected operations can support the reviewed languages.

2.1. Character String Operations

The UNIX[‡] operating system has an interesting history in that its origins were based on the interactive editing, documenting, and maintenance of programs that were to be submitted and executed on another system (Dolotta, Haight, and Mashey 1978). This orientation places considerable emphasis on non-numeric operations and especially on character string operations. A plethora of textually oriented tools and filters have grown with UNIX as it has expanded into the research and commercial marketplace. This makes a UNIX based system an excellent environment in which to examine the gains of hardware accelerated character string operations. The coprocessor design in this dissertation implements the set of character string operations defined by the UNIX operating system library of routines. However, we recognize that UNIX does not represent the entire world of computing and each UNIX system will have different hardware support depending on the CPU of the system. We therefore review a number of string oriented languages and CPU designs to determine general requirements for string operations and their traditional hardware support. To simplify this review and discuss how the coprocessor might support the languages, the operations are defined first and the review follows.

2.2. MEMORY and STRING

Appendix A includes the manual pages for a detailed description of the string operations which are being implemented. The operations associated with **STRING** are all based on the assumption that the strings are terminated with the null character and hence the operations know when to halt. The operations associated with **MEMORY** are supposed to be optimized for the CPU architecture and require a length parameter to indicate how long the strings are. There is an overlap in functionality in the definitions of these operations, but implementing all of them in hardware provides a wide base from which to support the higher level languages presented later.

The operations can be generalized into three categories.

(1) Copying - *memccpy, memcpy, memset, strcat, strncat, strcpy,* and *strncpy* are all operations which perform the task of copying a string from one place in memory to another place.

[‡] UNIX is a trademark of AT&T.

(2) Searching - *memchr, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok, index,* and *rindex* are all associated with finding a character or one of a set of characters in a string.

(3) Comparing - *memcmp, strcmp,* and *strncmp* are used to compare two strings lexicographically.

These operations are given the memory addresses of the first characters of the string operands and loop through these operands performing the appropriate operation. This library is somewhat different from many string operations in its use of a terminator character to denote the end of a string. However, several operations are provided that take a length parameter, thus performing the operations in a more traditional fashion.

Two terms associated with searching will be used quite often. These are *scan* and *span*. The *scan* operation searches for the first occurrence in one string of any character contained in another string. The *span* operation searches for the first position that does **not** contain any characters from the other string. These operations are useful for finding delimiters in strings and breaking out fields. For example, command processors could use sequences of *scans* and *spans* to parse a command line. The *strtok* operation combines *scans* and *spans* in just that fashion.

The copying and comparing categories can be implemented fairly easily in hardware. These appear in a variety of forms in the CPU designs that are reviewed. Searching is not as easily accomplished. As such, only limited searching capability is presented in the larger CPU designs reviewed. The review of the languages shows an intensive need for powerful searching capabilities which are not represented in our current set of operations. Chapter 3 will be dedicated to addressing searching algorithms and presents a hardware algorithm that can be incorporated into the coprocessor design.

2.3. String Languages Reviewed

The term "string languages" is used to loosely describe languages or applications which are predominately oriented towards string operations. In each subsection, the "string language" will be discussed in general terms and its requirements will be related to the set of operations defined in Appendix A.

2.3.1. LISP

LISP (Winston 1979; Winston and Horn 1981) is one of the oldest surviving, non-numeric programming languages. Unfortunately it is *symbolic* and *link* oriented rather than character string oriented. This makes it completely different from all the other non-numeric languages in this chapter. It is mentioned here for completeness in the review but it is inappropriate for our architecture. The reader is referred to Greenblatt (1980) for a hardware design which supports LISP.

2.3.2. SNOBOL

The original SNOBOL (Farber, Griswold, and Polonsky 1964) programming language was a very simple language with only one data type, the string, and few pattern matching statements expressed in a rigid syntax (Coutant, Griswold, and Wampler 1980). Various versions of SNOBOL were developed leading up to SNOBOL4 (Griswold, Poage, and Polonsky 1971) which introduced a variety of data types and the abilities to dynamically construct and manipulate patterns as data objects. While the first SNOBOL was dedicated to special-purpose string processing, SNOBOL4 could be considered more general-purpose and is in wide use (Griswold 1979).

The control structures of SNOBOL4 are based on the success or failure of pattern matching operations. Most of the extremely simple patterns and a majority of the functions could be built around the operations defined for our coprocessor with very little modification, but the searching operations defined for our coprocessor are entirely inadequate. This is unfortunate since the majority of the execution time of SNOBOL4 is spent in pattern matching. Because of the dynamics of building patterns on the fly and arbitrarily shifting pointers during the match, SNOBOL4 represents a special challenge in pattern matching (Gimpel 1973).

Chapter 3 of this dissertation presents an algorithm that was originally intended to be used with a hardware SNOBOL engine (Mukhopadhyay 1979). The algorithm is greatly expanded and can be effectively exploited by this and other string languages.

2.3.3. ICON

The ICON (Coutant, Griswold, and Wampler 1980) programming language is a direct descendent of the SNOBOL languages with some of the same authors. While the original SNOBOL was designed with the single string data type, ICON is intended to be a general-purpose programming language with an emphasis on string processing. ICON uses control structures and numeric math in a manner similar to ALGOL but augments this with string data types, character set data types, plus operators, functions, and type conversions which work on these new data types. If we were to choose a particular string language to support, rather than a general-purpose solution, this would be the language of choice.

The ICON compiler was written in the C programming language under the UNIX operating system and hence could probably make heavy use of the string library we are implementing. However, the ICON language itself is a higher level abstraction and relieves the programmer from the concerns of memory allocation and terminator characters, and also adds the character set (*cset*) data type which is only indirectly approached in our library searching routines.

Chapter 4 will address some special-purpose hardware which could significantly aid the *cset* operations. Furthermore, the searching operations as defined in Appendix A are once again totally inadequate for the searching requirements of ICON. These inadequacies will be overcome by the functionality added in Chapter 3.

2.3.4. AWK and SED

These cryptic names belong to two very powerful string languages found in the UNIX environment. AWK (Aho, Kernighan, and Weinberger 1978) accepts a string of input characters and breaks it into fields and records. Patterns are specified which are built from those fields and records. When the pattern is matched, a C language style routine associated with the pattern is executed.

SED (Kernighan and McIlroy 1978) is a stream editor which performs editing based on a series of line oriented commands. Each line of input is passed through the series of editing commands and manipulated appropriately.

Because both of these languages are associated with the UNIX environment, they consider strings to be null terminated and make direct use of the library routines of Appendix A. However, both of these languages also require the extensive use of complex searching as represented by the algorithms presented in Chapter 3.

2.3.5. Macro Preprocessors

While macro preprocessors might not be directly considered "string languages", they are unquestionably string oriented applications. The preprocessors reviewed are the PL/I (Hughes 1979), IBM OS (Vickers 1971), and C (Kernighan and Ritchie 1978) language macro preprocessors. All of these preprocessors have the common functions of performing file inclusion, macro expansion, and conditional compilation, all of which are accomplished through search and replace string operations. These would certainly benefit from the instruction set of our coprocessor combined with the searching algorithm of Chapter 3.

2.4. CPU Support of Character Strings Review

In order to review the industrial support of character strings in hardware, the instruction sets of a cross section of microprocessors, a mini-computer, and a main-frame are examined. The microprocessors are especially interesting since they have the knowledge of past designs to build from and they are growing their instruction sets with each new release.

2.4.1. 8-Bit Microprocessors

In the general purpose computer market of 8-bit microprocessors, three CPUs seem to appear more than any others. These are the Zilog Z-80 (1978), Intel 8080, and the Motorola MC6809 (1979). The instruction sets of the Intel 8080 and Motorola MC6809 have no instructions that could be considered more useful for character strings than numeric data. The Zilog Z-80, however, has a few instructions that are useful for character string manipulation. These instructions include a set of block moves for transferring several bytes of memory from one place to another, and a set of compare instructions that will search an area of memory and indicate if some memory word matched the contents of the accumulator. These instructions are able to move through the memory references by increasing or decreasing addresses. This provides the basis of functionality for some of our copy and search operations, but not all of them. It also provides functionality which we do not provide, specifically the ability to work through the strings backwards.

It would be fair to mention that these 8-bit microprocessors made a tremendous impact on the world of computing, but due to technological limitations of the time, they were limited by the amount of computing power that could be placed in them. Of the instruction set of these CPUs, only the MC6809 has a multiply, and it is for unsigned integers only. These processors are generally boosted in operating power by the addition of peripheral processors. Our coprocessor design could have provided precisely that boost for string operations.

2.4.2. 16-Bit Microprocessors

The 16-bit versions of the three popular 8-bit microprocessors, the Zilog Z-8000 (1979), the Intel 8086 (Rector and Alexy 1980), and the Motorola MC68000 (1980), are also the most popular on the market. Some maturing of instruction sets is evident. These include multiplies, divides, and most notably, extensive support for operating systems. However, character strings still do not seem to have been recognized as an important data type. The MC68000 contains no instructions that support characters as anything more than integer values. The Intel 8086 has a set of instructions that it calls string primitive instructions, but in fact these instructions only work with 8-bits or 16-bits which can represent only one or two characters at a time. These instructions can be repeated until a register that is always altered by the instructions attains a certain value. The Intel 8086 character string capabilities are roughly comparable to the 8-bit Z-80. The Z-8000 maintained the character string capabilities of the Z-80 and added a translation capability enabling support of some high level language format print and conversion statements.

Another 16-bit microprocessor that has some degree of character string instructions is the National Semiconductor NS16000 (1981; Bal 1980a; Bal 1980b; O'Dowd, Kohn, and Soha 1980). This processor has been credited with providing clever support for the operating system environment. The character string instructions are similar to the Z-8000 in that there are move, compare, and translate commands that can traverse both directions through the string. Both processors require that the length of the strings already be known by the programmer and loaded into specific registers.

Another example of a processor that is made to support operating systems and high level languages is the Western Electric BELLMAC-32 (Berenbaum, Condry, and Lu 1982). The two string operations that are present are direct implementations of the *strcpy* and *strlen* operations of our library. This is not coincidental since the intent of the design of the WE32000 is to support UNIX. The string terminator concept is not usually found in a CPU design since these operations can get carried away if the string is not properly terminated. The string terminator is a good idea, though, because it relieves the programmer from the responsibility of keeping track of the string length. On the other hand, some overhead is induced by not knowing the string length, especially if we wish to work from the end of the string (e.g., concatenation or a compare starting at the end of string and working towards the beginning).

2.4.3. 32-Bit Microprocessors

The 16-bit microprocessors basically performed 32-bit operations but had a 16-bit data path due to various constraints. As the constraints were overcome, those processors were able to incorporate a full 32-bit data path and expand their addressing capabilities. An interesting growth path to follow is the Motorola chip as it moved from the MC68000 to the MC68010 (1983) to the MC68020 (1984) to the MC68030. Among other innovations, the MC68010 incorporated the ability to hold a small, tight loop in a three word instruction cache, thus reducing the instruction fetch and instruction decode overhead. This could actually be enough to implement a small subset of the string instructions in our library. The MC68020 expanded this to 128 words of on-chip instruction cache and added a series of coprocessor interface instructions. The MC68030 incorporates an MMU into the CPU design adding still more non-numeric functionality to the central processor.

2.4.4. DEC VAX 11/780 Mini-Computer

While the microprocessors have been frugal in their acceptance of string operations, the larger CPUs can afford the flexibility to include some character string operations in their instruction sets. The two non-VLSI CPU instruction sets that are reviewed are the DEC VAX 11/780 and the IBM SYSTEM/370. Both have an interesting set of character string operations.

The character string operations of the VAX (DEC 1977) instruction set include block move, string translation, string comparison, and operations that identify positions of interest in a character string. It is also not coincidental that these operations will map closely to the instruction set of our coprocessor since the UNIX library was influenced by the VAX instruction set and attempted to exploit the CPU instructions to their fullest.

The block move is no different from that on any of the other machines. The translation operations are slightly different from most, though. The usual translation operation is to pass through a string, replacing characters with specified replacement characters. The VAX does this too, but the translation is done during a block move. This means the operation can be kept from doing a destructive translation to the source string. Furthermore, there is the option of having the translation stop when a specific escape character is encountered. The compare is not different from previously described compares, but limits the order of the comparison from the start to the end of the string. The real interesting operations are the positioning operations. The LOCC (LoCate Character) operation returns the first position in the string at which a specified character appears. The SKP (SKip character) operation returns the first position in the string at which a specified character does *not* appear. The SCANC (SCAN Characters) and SPANC (SPAN Characters) operations perform the same functions as LOCC and SKP, respectively, except a set of characters is compared rather than one specific character. The MATCHC (MATCH Characters) operation returns the first occurrence of a specified string in the object string (position of a substring).

The LOCC, SKP, SCANC, and SPANC all have direct mappings to operations in our coprocessor, but we will implement them differently from the VAX. The functionality of MATCHC is also provided in Chapter 3, but using a different algorithm.

2.4.5. IBM SYSTEM/370

The 370 instruction set (Vickers 1971) includes some interesting string operations. Here the character is considered a data type and the instruction set allows for manipulations in registers and memory, recognizes that one character string might exist across several memory words, and provides conversion operations between the character form of numbers and an internal format. The instructions can be separated into 5 categories: comparison, character movement, string movement, translation, and conversion.

The comparison operations allow you to compare two strings in memory up to a length of 256 characters (CLC), to compare a character in memory with an immediate operand (CLI), to compare parts of a register with words in memory (CLM), or to compare two very long strings (longer than 256 characters) stored in memory (CLCL).

The Insert Character (IC) and Insert Character Under Mask (ICM) instructions will load characters from memory into registers. The Store Character (STC) and Store Character Under Mask (STCM) instructions will copy characters from registers into memory. The "Mask" instructions are necessary because characters only use 8 bits on IBM machines and the words and registers are all 32 bits long. The masks are used to specify which 8-bits (byte) of the 32 bits (4 bytes) are to be altered.

The string movement operations cover the movement of one character to memory, the movement of one string of length up to 256 to another memory location, and the movement of one string of length up to 16 million characters to another memory location.

The translate command will perform the usual pass through a string, replacing characters to their translated value. The TRanslate and Test (TRT) is an interesting operation. In this case no translation is actually performed, but the position of the first character in the string that had a non-zero entry for the translation is returned. This is a way to perform the SCANC function described under the DEC VAX 11/780 and is in fact the SCAN pattern matching function of SNOBOL4.

The PACK and UNPK (UN PacK) commands convert between the "packed decimal" internal format (compacted BCD) and the character format for numbers. Other instructions can perform math on the packed decimal numbers. The most complicated string operations available on the 370 are related to this conversion and translation. These instructions are the ED (EDit) and EDMK (EDit and MarK). These instructions essentially perform the picture formats of COBOL and PL/I. The EDit operation converts a packed decimal number to character format, but it can suppress leading zeros on the number by converting them to blanks. The Edit and Mark will perform the same function but can fill the leading insignificant zero digits with character patterns, such as dollar signs or asterisks.

This machine shows the strongest recognition of character strings as a data type, but the manipulations primarily provide means to cope with individual bytes in a 32 bit word environment. The most sophisticated instructions are specifically written to support the format conversions for high level languages.

2.5. Conclusions on the Selection of Operations

The review of the string languages indicates that a much stronger searching capability needs to be present than is represented in the MEMORY and STRING set of operations. The UNIX library includes a routine for the compilation of a form of a regular expression (*regexp*) which is used by many of the string applications such as *awk* and *sed*. The next chapter is dedicated to the problems of searching. It provides a design which incorporates full regular expressions and is thus sufficiently powerful to support any of the languages discussed here.

Some of the CPU instruction sets discussed here added the ability to traverse strings through incrementing or decrementing loops. The UNIX string library would not make direct use of this feature since it uses the terminator character at the end of the string to determine when to halt execution of most operations. The single operation that comes close to needing decrementing loops is *strrchr* which looks for the first occurrence of a character from the end of the string. However, *strrchr* can be (and is) implemented through a single forward pass. Any operations that use a counter instead of the terminator are trivial to implement in either direction.

In examining the performance increase of the selected set of operations, we can determine what types of gains can be expected through simple firmware micro-coding of the copy and compare operations and we can demonstrate the advantages that can be had through the use of special hardware for searching.

CHAPTER 3

SEARCHING

As emphasized in the previous chapter, *searching* and *pattern matching* play a major role in non-numeric processing. The most general definition of a searching algorithm is defined as follows:

> Let there be a pattern called P and some amount of data called D. The algorithm will find all occurrences of P in D.

No assumptions are made about D. The data is not sorted, indexed, or blocked in any fashion known to the algorithm, nor is the type of data defined. It may consist of byte length characters, word length integers, double word reals, or even a complex structure of mixed data types. The only restriction is that each element of the data must be of uniform size.

The power and flexibility available in specifying the pattern *(P)* varies from one searching algorithm to the next. A few algorithms (Boyer and Moore 1977; Galil and Seiferas 1983; Galil 1984; Knuth, Morris, and Pratt 1977) restrict the pattern specification to consist exclusively of a sequence of data elements. Some algorithms (Aho and Corasick 1975; Bird 1979) add the ability to search for multiple patterns simultaneously. Other algorithms introduce wild card characters (Curry and Mukhopadhyay 1983; Fischer and Paterson 1974; Foster and Kung 1980; Mukhopadhyay 1979; Roberts 1977) which are single characters representing the entire alphabet. The most powerful pattern specification algorithms (Floyd and Ullman 1980; Foster and Kung 1981; Haskin 1980; Lee 1986; Thompson 1968; Trickey 1982) use regular expressions to describe the pattern.

In addition to this wide range of pattern specifications, the algorithms differ with regard to other attributes as well. When attempting to decide on an appropriate algorithm, numerous factors come into play. These factors include the order in which the data is accessed (sequential or random access requirements), the time and space complexities for processing the pattern and performing the search, the operators allowed in the pattern specification, and the flexibility for redefining the data element size.

In software searching algorithms, extremely clever techniques have been employed to produce linear time and linear space complexities for *exact pattern matching* (i.e., pattern specifications with no wild card characters or expression operators). The introduction of wild card characters into the pattern specification immediately moves the software algorithms into nonlinear complexity (Fischer and Paterson 1974) and the use of regular expression operators in the pattern specification can result in exponential algorithm complexities (Aho and Corasick 1975; Floyd and Ullman 1980). Some hardware searching algorithms (Curry and Mukhopadhyay 1983; Foster and Kung 1980; Mukhopadhyay 1979) have an advantage over their software counterparts in their ability to search for patterns containing wild card characters while maintaining linear time and linear space complexity. The hardware algorithms which search with regular expression specifications vary from polynomial to exponential in time and space complexities. Unfortunately, the polynomial algorithms (Foster and Kung 1981; Floyd and Ullman 1980; Trickey 1982) have hardwired a fixed pattern into the design thus limiting the hardware to a one time definition of the pattern.

This chapter introduces a hardware algorithm which:

(1) Accesses the data *(D)* sequentially with no backtracking.

(2) Supports regular expression operators and wild card characters.

(3) Preprocesses the pattern in linear time and polynomial (quadratic) space.

(4) Searches in linear time.

(5) Is fully reprogrammable for new patterns and varying data element sizes.

At present, this is the only algorithm able to claim all of these attributes.

The remainder of this chapter will introduce general searching concepts and their relationship to formal language and set theory; define, prove, and refine variations of the algorithm for different pattern operations; present a VLSI implementation of one of those variations along with its fabrication results; examine some design alternatives; then conclude with a comparison of the algorithm to other software and hardware searching algorithms based on a variety of criteria.

3.1. Concepts in Searching

Both language and set theory have been studied and applied to computer science for over 40 years (Hopcroft and Ullman 1979). A searching algorithm can be directly developed out of traditional language theory techniques, and the notation and terminology of language theory can provide a common basis on which to compare different searching algorithms. This section will define the terminology to be used in this chapter and present a searching algorithm using language theory constructs.

The previous section stated that the most powerful searching algorithms use regular expressions to describe the pattern *(P)* which is to be searched for in the data *(D).* We can go on to state that all of the less powerful algorithms have patterns that are fully contained subsets of regular expressions and can be defined by placing certain restrictions on the operators allowed in the expression. A regular expression denotes a set of strings that are built from the characters of a finite alphabet. The set of strings defined by a regular expression can include the three operators, concatenation, alternation, and closure, combined with parentheses to clarify or override precedence. Regular expressions are formally defined in (Barrett and Couch 1979) by:

(1) Let Σ be a finite alphabet.

(2) Elements of Σ are regular expressions. For $a \in \Sigma$, the regular expression a denotes the set {a}.

(3) *Concatenation* is an associative, noncommutative binary operation. The token for concatenation is juxtaposition, e.g., if *E1* and *E2* are two regular expressions, then *E1E2* is the concatenation of the two. If *E1* and *E2* are two regular expressions denoting the sets of strings S_1 and S_2, respectively, then *E1E2* is a regular expression which denotes the set

$$S = \{uv \mid u \in S_1 \text{ and } v \in S_2\}$$

(4) *Alternation* is an associative, commutative binary operation, represented by the vertical bar symbol (|). If *E1* and *E2* are two regular expressions denoting the sets of strings S_1 and S_2, then *E1* | *E2* is a regular expression denoting $S_1 \cup S_2$ (the union of S_1 and S_2).

(5) *Closure* is a unary operation represented by an asterisk (*). If E is a regular expression denoting some set of strings S, then E^* is a regular expression called the *closure* of E and represents the set of all possible strings formed by choosing members of S and concatenating them. The empty string ε is also a member of the set. Thus the closure of a regular expression E is a compact way of writing the infinitely large regular expression

$$\varepsilon \mid E \mid EE \mid EEE \mid EEEE \mid \bullet\bullet\bullet$$

(6) Parentheses may be used to override or insure precedence. The default order of precedence from highest to lowest is closure, concatenation, alternation.

An example of a regular expression is:

((2|02)/(2|02)/)|((*Feb*|*February*) (2|02),)(80|1980)

This regular expression represents sixteen different ways to specify Ground Hog Day in 1980. Expanded, the set of patterns that would match this regular expression is:

P = {	"2/2/80",	"2/2/1980",	"2/02/80",	"2/02/1980",
	"02/2/80",	"02/2/1980",	"02/02/80",	"02/02/1980",
	"Feb 2, 80",	"Feb 2, 1980",	"Feb 02, 80",	"Feb 02, 1980",
	"February 2, 80",	"February 2, 1980",	"February 02, 80",	"February 02, 1980" }

The equivalence of regular expressions to regular languages and the languages accepted by finite state automata is well known in computer science (Barrett and Couch 1979; Harrison 1978; Hopcroft and Ullman 1979; Salomaa 1969) and algorithms exist to convert from one form to another. A *finite state automaton* is formally defined in (Barrett and Couch 1979) by a five-tuple $(Q, \Sigma, \delta, q_0, F)$, where

(1) Q is a finite set of states.

(2) Σ is a finite set of permissible input tokens, i.e., the alphabet of the language.

(3) δ is a partial function that maps a state and an input symbol to another state. δ is called the *state transition function*.

(4) q_0 is a designated state in Q called the initial or start state of the FSA.

(5) F is a subset of Q consisting of one or more accepting states.

The FSA is initially in the start state denoted by q_0. It operates through a sequence of moves to other states in Q where each move is defined by the present state and the next input character (an element of Σ) as defined by δ. If the FSA is left in a state contained in F when all input is exhausted, then the input data is accepted as a member of the language recognized by the FSA.

Consider an example where the alphabet is defined as $\Sigma = \{a, c, o\}$ and the entire language consists of the single word *cocoa*. An FSA which would determine if an input string of characters was in our language is defined by:

$$Q = \{S, A, B, C, D, E, G\}$$

$$\Sigma = \{a, c, o\}$$

$$\delta = \{\ \delta(S,c)=A,\ \delta(A,o)=B,\ \delta(B,c)=C,\ \delta(C,o)=D,\ \delta(D,a)=G,$$

$$\delta(S,\mathrm{a})=\delta(S,\mathrm{o})=\delta(A,\mathrm{a})=\delta(A,\mathrm{c})=\delta(B,\mathrm{a})=\delta(B,\mathrm{o})=\delta(C,\mathrm{a})=\delta(C,\mathrm{c})=\delta(D,\mathrm{c})=\delta(D,\mathrm{o})=$$
$$\delta(G,\mathrm{a})=\delta(G,\mathrm{c})=\delta(G,\mathrm{o})=\delta(E,\mathrm{a})=\delta(E,\mathrm{c})=\delta(E,\mathrm{o})=E\ \}$$

$$q_0 = \{\ S\ \}$$

$$F = \{\ G\ \}$$

State S is the start state, G is the accepting state, and E is a special error state. All other states are intermediate steps towards E or G. Once the FSA enters the error state, it never leaves and the input string cannot be accepted. This FSA would be called a *completely specified, deterministic FSA*. An FSA is said to be *completely specified* if there is a transition defined in δ for every input character for every state (i.e. δ is a function). An FSA is said to be *deterministic* if each of those state transitions is unique. An FSA is said to be *non-deterministic* if there exists two or more transitions for the same input symbol in the same state (i.e. If δ is a mapping of a state and an input symbol into the subsets of the state set, then the FSA is called non-deterministic).

Two other forms of representing an FSA are popular. The first form is that of a table which can be easily implemented in a computer program and the second is a state diagram which provides an easy visual representation for humans. Our *cocoa* example in tabular form would be:

TABLE 3.1. Tabular Form of 'cocoa' FSA

	Input Character		
Current State	a	c	o
S	E	A	E
A	E	E	B
B	E	C	E
C	E	E	D
D	G	E	E
E	E	E	E
G	E	E	E

When a new input character is received, the column entry associated with that character in the row of the current state represents the new state. Similarly, the same example in a state diagram is shown here:

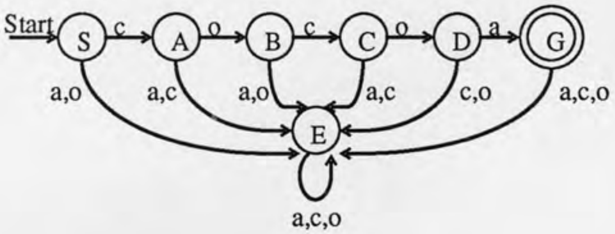


Figure 3.1. FSA for Parsing 'cocoa'

In this diagram, the circles represent the states and the arcs represent the transitions from one state to the next. Each arc is labeled with the character or set of characters that causes the transition of states. The double circle around state *F* denotes that it is an accepting state.

A recursive technique is defined in (Barrett and Couch 1979) which converts any arbitrary regular expression to an equivalent FSA state diagram. In this technique, state *S* is defined as a start state, state *F* is defined as a final state, *E* in a box represents an expression that still requires refinement, *E1* and *E2* represent subexpressions that can be independent expressions, and ε represents a transition that can occur without an input character (called an *ε-move*). The following seven rules convert any arbitrary regular expression to an equivalent FSA state diagram.

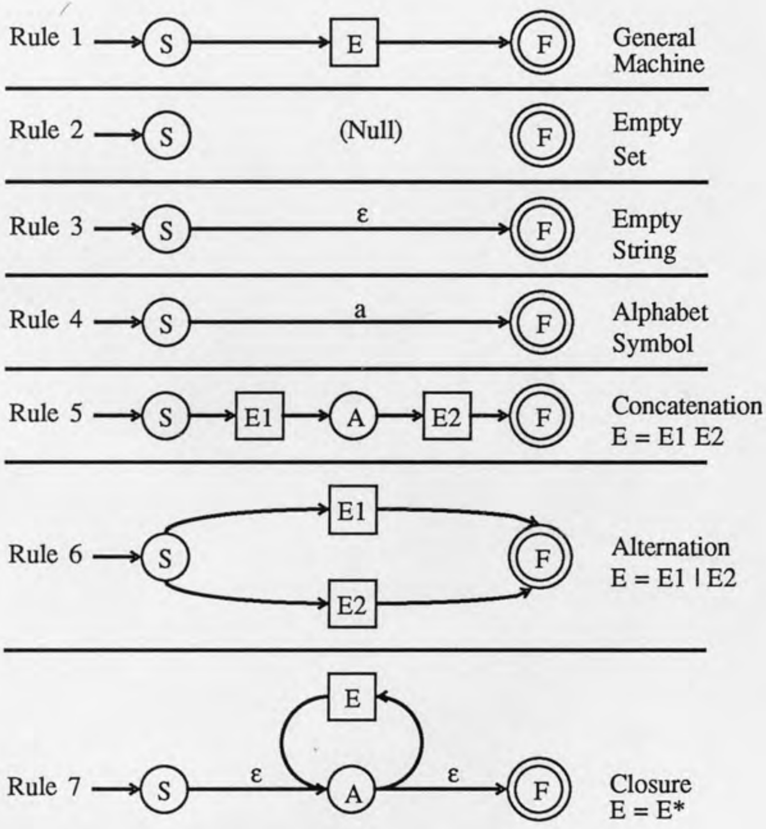


Figure 3.2. Barrett and Couch Illustration for NFSA Construction

Applying these rules to any arbitrary regular expression yields an incompletely specified FSA state diagram. Consider the *cocoa* example from earlier in this section. The following diagram illustrates the application of the first few rules to the regular expression *cocoa* and then shows the final result. The intermediate states are denoted as *A*, *B*, *C*, and *D*.

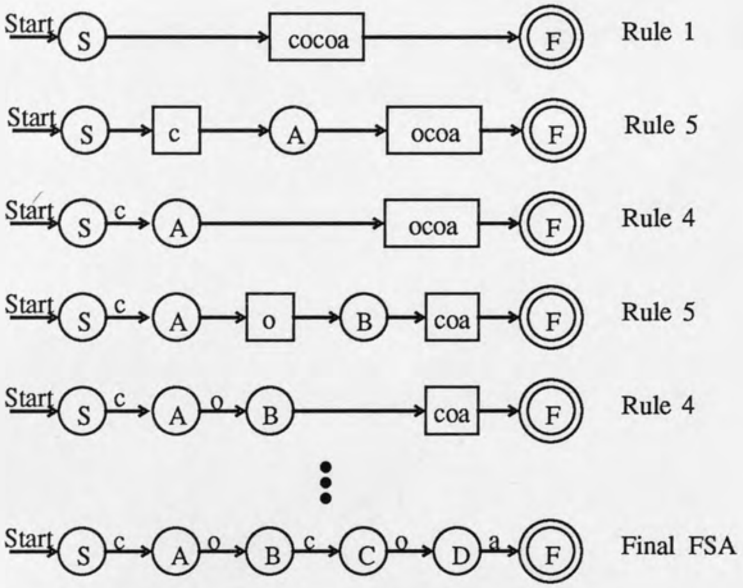


Figure 3.3. Application of Recursive Rules to 'cocoa'

This FSA is quite similar to the one of Figure 3.1 with one distinction, the lack of the error state and its associated transitions. This is trivially resolved since every transition not specified in the final FSA of Figure 3.3 is an error transition and can be easily specified as a transition to an error state. This is perhaps easier to visualize in the tabular form.

TABLE 3.2. Tabular Form of the 'cocoa' Incomplete FSA

	Input Character		
Current State	a	c	o
S	-	A	-
A	-	-	B
B	-	C	-
C	-	-	D
D	F	-	-
F	-	-	-

If every "-" entry of this table is replaced with a transition to a new error state, then this incompletely specified FSA becomes a completely specified parser FSA. In fact, it is the general case that the FSA generated by the rules of Figure 3.2 will only be incomplete by the error transitions, and therefore a parser can always be trivially generated from the incomplete FSA.

If instead we would prefer to make the incomplete FSA *search* rather than *parse*, a different change can be made. Consider the functions of a parser and a searcher. A parser is given the task to determine if an input stream of characters is a proper member of the language. As soon as something goes wrong, it can enter and remain in an error state (we are, of course, discussing the pure concept of parsing and avoiding the practical and generally necessary aspects of error recovery and continuation). Therefore, the addition of the error state and its transitions are all that is necessary to turn the incompletely defined FSA into a parser. A searcher on the other hand, is given the task to find members of its language in the midst of a stream of data. There is no need for an error state and any undefined transition that is encountered can simply be discarded since that would imply a failure to match the pattern. Upon reaching an accepting state, a match can be announced. Because each new input character could be the start of a possible match, the only change required to the incompletely specified FSA is a transition from the start state back to the start state for every character in the alphabet as shown here for the *cocoa* example.

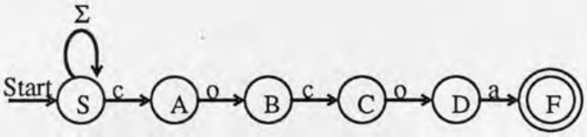


Figure 3.4. NFSA for ‘cocoa’

This is labeled as an NFSA standing for Non-deterministic Finite State Automaton. Since the start state has a transition both to itself and to state *A* on the same character, the FSA is non-deterministic. Algorithms are known (Barrett and Couch 1979; Harrison 1978; Hopcroft and Ullman 1979; Salomaa 1969) to eliminate all transitions on no input (ε-*moves*) and convert from non-deterministic to deterministic. If these algorithms were applied to the NFSA of Figure 3.4, the resulting DFSA (deterministic FSA) would be:

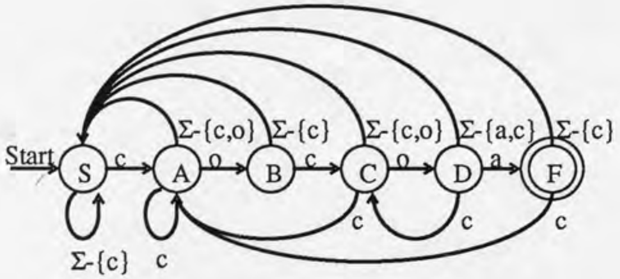


Figure 3.5. DFSA for Searching 'cocoa'

Note that this DFSA for searching is dramatically different from the DFSA for parsing shown in Figure 3.1 but both can be derived from the incompletely defined FSA in Figure 3.3.

In a uniprocessor software environment, it is quite difficult to implement a non-deterministic algorithm efficiently. Therefore the conversion to a DFSA is invariably performed. Unfortunately, the algorithms which transform an NFSA to a DFSA can explode the number of states to exponential size (i.e., if the NFSA had *n* states, the DFSA could have 2^n states).

If the original regular expression describing the pattern is restricted to the concatenation operation (i.e., containing only elements of Σ), then the pattern is said to be an *exact* or *fixed* pattern and the final DFSA can be guaranteed to have a linear number of states related to the length of the pattern (Knuth, Morris, and Pratt 1977). Allowing alternation but no closure or parentheses can still maintain linearity (Aho and Corasick 1975). Relaxing any more restrictions on the regular expression can no longer guarantee linearity (Fischer and Paterson 1974).

Some hardware searching algorithms immediately realize an advantage by keeping the FSA in its non-deterministic form and thus avoiding the exponential expansion. This is accomplished through the use of multiple processors which can allow multiple states to be active simultaneously. Furthermore, because the goal is to *search* rather than *parse*, certain assumptions can be made about the original regular expression. Searching for the *empty set* is meaningless since that would be searching for nothing. Thus rule 2 of Figure 3.2 is unnecessary for a searching algorithm. Rule 3 is also unnecessary if ε-*moves* are not allowed in the regular expression. This does not limit the power of the algorithm since it can be shown (Barrett and Couch 1979; Harrison

1978; Hopcroft and Ullman 1979; Salomaa 1969) that the set of languages recognized with and without ε-*moves* is equivalent. For example, the Ground Hog Day regular expression example used earlier has a subexpression "(2|02)" in it several times. This could also be expressed as "((0|ε)2)" stating the "2" can be preceded by a "0" or nothing. Both subexpressions are equivalent. If a new closure rule is introduced that does not use ε-*moves*, then all ε-*move* preprocessing is totally eliminated. To that end, the following set of rules is now presented for generating an incomplete NFSA from an arbitrary regular expression defining a search pattern.

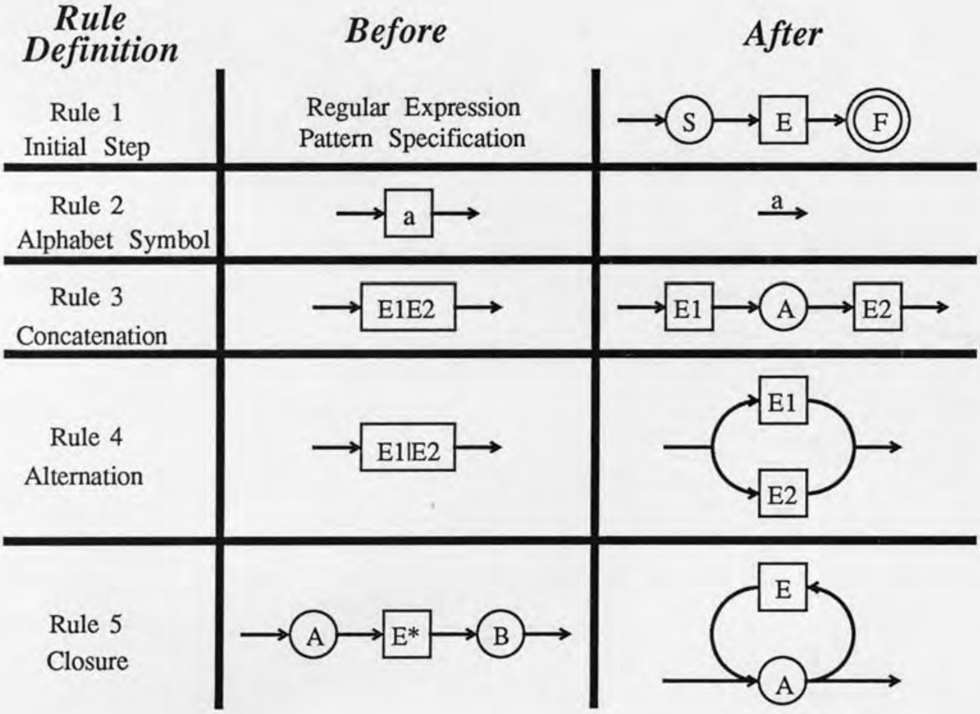


Figure 3.6. Final Regular Expression to FSA Rules

Not stated explicitly in the diagram for rule 5 is the additional constraint that the resulting state "A" in rule 5 inherits all the outputs and attributes of state "B" including changing "A" to an accepting state if "B" was one. Consider the example pattern *P* = *"a(b|c)*"* and apply these rules.

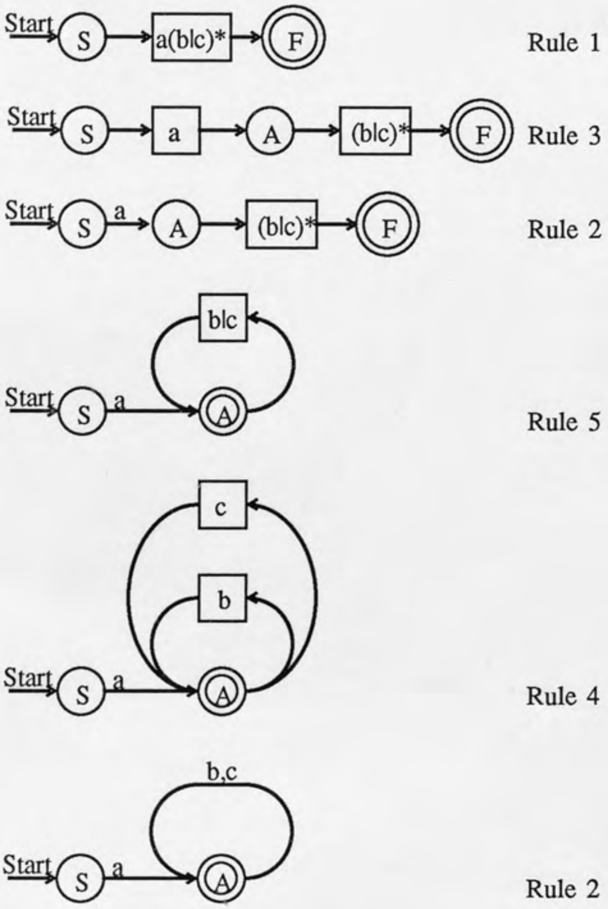


Figure 3.7. FSA for 'a(b|c)*'

This pattern will match the string "a" and any string that starts with an "a" which is followed by any number and any combination of "b"s or "c"s. When rule 5 was applied, state *A* acquired the accepting state attribute of *F* and would have acquired all the outputs of *F* if there had been any.

The operators and terminology of regular expressions will now be used to specify the patterns of different searching algorithms. The rules of Figure 3.6 will then be applied to create an incomplete NFSA which will then have the transition on Σ, from and back to the start state, added. This final NFSA is then implemented in a hardware algorithm.

3.2. The Hardware Algorithm

Because of the complexity of the hardware and operations, it is convenient to start with a simple subset and build up to the complete algorithm. To that end, the algorithm is presented in the following stages:

(1) Exact pattern matching.

(2) Simultaneous search for multiple exact patterns.

(3) Wild card characters.

(4) Full regular expression searching.

These stages also provide a convenient mapping to the various capabilities of other algorithms against which this algorithm will later be compared.

3.2.1. Exact Pattern Matching

As defined earlier, an *exact* or *fixed* pattern is one in which only elements of the alphabet (Σ) appear in the pattern. There are no alternation or closure operators and concatenation is directly implied by the juxtaposition of characters from Σ.

The following hardware cell is designed to implement an NFSA state and its associated transition.

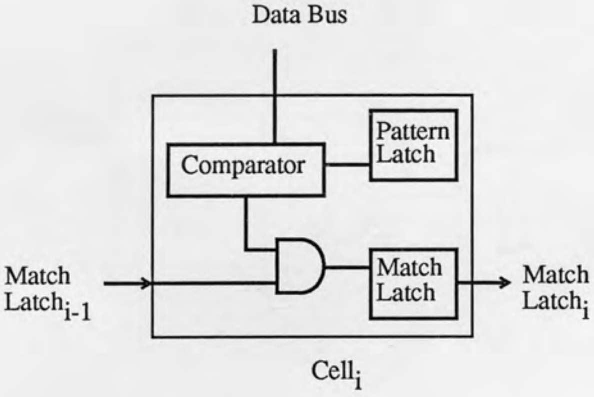


Figure 3.8. Single Exact Pattern Cell

The *Data Bus* provides the current input character to the cell. The *Pattern Latch* is a latch which holds a single character of the pattern. *Comparator* logic yields a 1 if the current input character is the same as the contents of the *Pattern Latch* and yields a 0 otherwise. *Match Latch*$_{i-1}$ represents the condition of any states which are a prefix to this state and *Match Latch*$_i$ provides the condition information of this state to any suffix states. The latch labeled *Match Latch* serves to delay the passing of the state information to occur simultaneously with the next input character.

A series of these cells interconnected is capable of searching for an exact pattern. This design could be considered a direct mapping to a state diagram of an NFSA as follows:

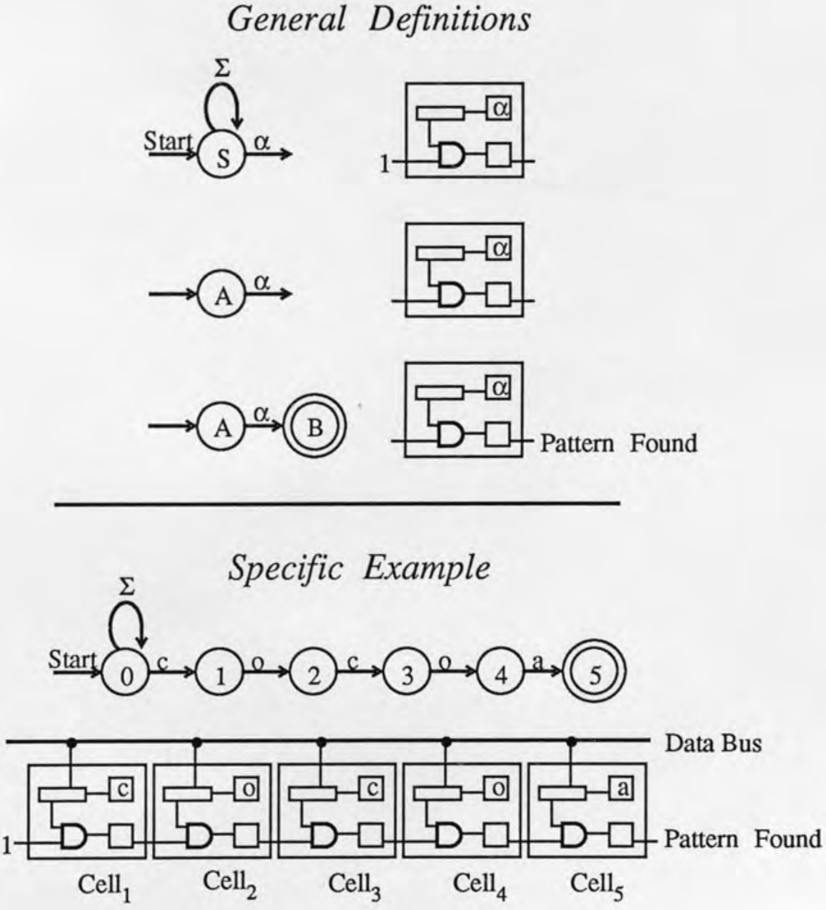


Figure 3.9. Mapping of FSA to Hardware Cell

Each *Pattern Latch* of a cell holds the transition character of the NFSA and each *Match Latch*$_i$ output is the current state of the automaton. State 0, being the start state of the NFSA and the state which is entered for every input character, is represented by the forced high input to *Cell*$_1$. A 1 output from *Cell*$_5$ (or in the general case, the last cell) is equivalent to a transition to the NFSA state 5 which is the accepting state and announces a match of the pattern in the input stream. Since multiple *Match Latch*$_i$ outputs can be high simultaneously, the non-determinism is handled directly by the hardware.

To illustrate how the non-determinism is managed, consider the input stream "cococoa" applied to the NFSA and hardware of the specific example in Figure 3.9.

TABLE 3.3. Trace of Input String 'cococoa'

Input Character	NFSA States Active	*Match Latch*$_i$ with value = 1
c	1	1
o	2	2
c	1,3	1,3
o	2,4	2,4
c	1,3	1,3
o	2,4	2,4
a	5	5

There is a direct one-to-one mapping of NFSA arcs to *Pattern Latch* values and NFSA states to *Match Latch* values with no superfluous hardware. The concatenation operation (and coincidentally, exact pattern matching) is therefore proven correctly implemented through a one-to-one mapping to an NFSA. Unfortunately, this algorithm lacks practical application because it requires exactly the same number of cells as the length of the pattern and is therefore somewhat inflexible. This problem is resolved in the next section.

3.2.2. Multiple Exact Patterns

If a requirement existed to search for several different patterns in the same data stream, it would be quite inefficient to load a single pattern, search the data, load the next pattern and search the data again, etc. Instead, it is desirable to be able to search for all of the different patterns in one pass of the data. Linearity can still be maintained in a uniprocessor software algorithm (Aho and Corasick 1975) and only a minor change to the hardware algorithm is required to accomplish this same goal.

A simple means to specify multiple exact patterns is through the use of the alternation operator. Note that parentheses are not yet allowed in the pattern, thus restricting the alternation to entire patterns. An example might be a search for both the words "cab" and "cat". Each is an exact pattern on its own but both could be joined through the alternation operation to be considered as one pattern consisting of two subpatterns. The new example pattern would then be "cab|cat". Converting from the regular expression to an NFSA and then directly extrapolating to the hardware exact-pattern-algorithm would yield the following results. Note only the *Pattern Latch* contents are displayed in the hardware cells. All other hardware is identical and therefore unnecessary to the diagram.

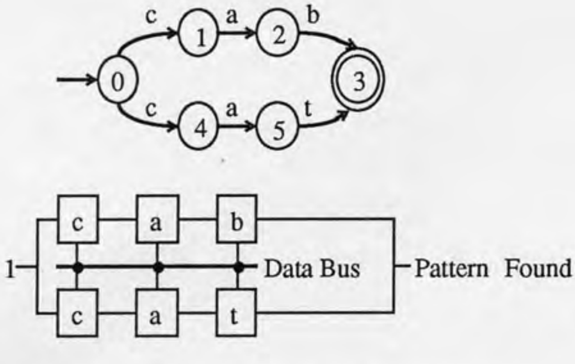


Figure 3.10. NFSA and Hardware for 'cab|cat'

While this is certainly correct, it lacks flexibility. Each subpattern is still required to have exactly the correct number of cells and special external connections are required to several cells. These restrictions can be eliminated by the addition of a small amount of hardware to the basic cell.

The *Match Latch*$_{i-1}$ input to each cell that starts a subpattern must always be a 1. The *Match Latch*$_i$ output of each cell that ends a subpattern must be made externally available to recognize that a match has been found. If each cell is provided with the knowledge of whether or not it is on a subpattern boundary, then all of the patterns can be arranged adjacently in a linear array of cells. The *EOP Latch* (End Of Pattern Latch) and its associated logic are added to accomplish this as shown in Figure 3.11.

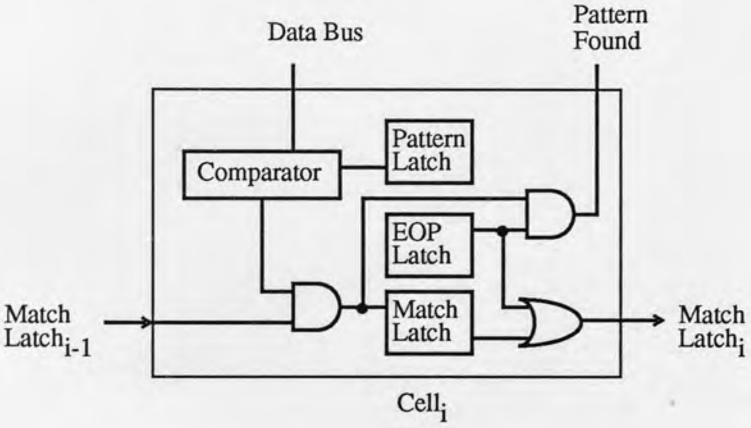


Figure 3.11. EOP Addition to the Basic Cell

If a 0 is loaded into the *EOP Latch*, then the logic of the cell is unchanged from the *Exact Pattern Cell.* If a 1 is placed in the *EOP Latch*, then the cell marks the end of one subpattern and the next cell can be the start of another subpattern. The *and-gate* logic out of the *EOP Latch* insures that only entire patterns matched provide input to the "Pattern Found" logic and the *or-gate* logic forces a continuous 1 input to the start of another subpattern.

The "cab|cat" example is shown in Figure 3.12 with only the *Pattern Latch* and *EOP Latch* values listed in the cells. Two extra cells are shown to illustrate that the restriction of an exact number of cells to pattern characters is eliminated. Furthermore, the first cell still has a 1 forced on its *Match Latch*$_{i-1}$ input but no other special connections are required. The "Pattern Found" logic can be as simple as a single signal indicating a match or as sophisticated as announcing exactly which cell matched the pattern and hence which subpattern.

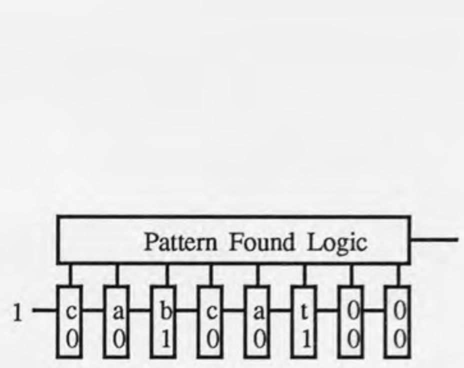


Figure 3.12. Hardware for ‘cab|cat’

A table tracing the NFSA of Figure 3.10 and the hardware cells of Figure 3.12 on the string "cab" illustrates that the one-to-one mapping still holds if the "Pattern Found" logic is considered equivalent to an accepting state.

TABLE 3.4. Trace of Input String ‘cab’

Input Character	NFSA States Active	*Match Latch*$_i$ with value = 1
c	1,4	1,4
a	2,5	2,5
b	3	3

The recursive rules defined in Figure 3.6 for converting a regular expression to an NFSA provide a convenient means for generating a hardware instantiation of an expression, but it is difficult to quantify certain attributes of the algorithm. The following algorithm can be used to *preprocess* the pattern thereby loading the hardware appropriately. From this, a quantification of the number of cells required and the time to process the pattern can be determined. One subroutine called *shift_pattern_in()* is used but not defined in this preprocessing algorithm. The sole purpose of this subroutine is to simultaneously shift the contents of all cells into their immediately adjacent cell to the right. The first cell will have the contents of the *Pattern* and *EOP* variables shifted into it. The variable *P* is the regular expression (pattern).

```
pat_ptr = length(P);
alternation_flag = 1;
while (pat_ptr > 0) do
begin
  Pattern = EOP = 0;
  switch (P[pat_ptr])
  begin
       case '|':
         alternation_flag = 1;
         break;
       default:
         Pattern = P[pat_ptr];
         if (alternation_flag) then do
         begin
              EOP = 1;
              alternation_flag = 0;
         end;
         shift_pattern_in();
  end;
  pat_ptr = pat_ptr - 1;
end;
```

This algorithm starts at the end of the pattern *(P)* and "shifts" the appropriate values for the *Pattern Latch* and *EOP Latch* into the cells. The *pat_ptr* variable is used to move through *P*. If an alternation operator is encountered (or this is the end of the entire pattern), then the *EOP Latch* should be set to 1. If the hardware consisted of eight cleared cells and this algorithm was applied to the "cab|cat" example, the following would occur.

TABLE 3.5. Trace of Preprocessing 'cab|cat' Pattern

<table>
<tr><th>Pattern
pat_ptr</th><th>Time</th><th colspan="8"><i>Pattern Latch</i> Values
<i>EOP Latch</i> Values</th><th>Comments</th></tr>
<tr><td>cab|cat
      ^</td><td>0</td><td>0
0</td><td>0
0</td><td>0
0</td><td>0
0</td><td>0
0</td><td>0
0</td><td>0
0</td><td>0
0</td><td>Initially all cells empty.
<i>pat_ptr</i> is at end of pattern</td></tr>
<tr><td>cab|cat
      ^</td><td>1</td><td>t
1</td><td>0
0</td><td>0
0</td><td>0
0</td><td>0
0</td><td>0
0</td><td>0
0</td><td>0
0</td><td>Last char of P shifted in.
At the end of a subpattern.</td></tr>
<tr><td>cab|cat
     ^</td><td>2</td><td>a
0</td><td>t
1</td><td>0
0</td><td>0
0</td><td>0
0</td><td>0
0</td><td>0
0</td><td>0
0</td><td></td></tr>
<tr><td>cab|cat
    ^</td><td>3</td><td>c
0</td><td>a
0</td><td>t
1</td><td>0
0</td><td>0
0</td><td>0
0</td><td>0
0</td><td>0
0</td><td></td></tr>
<tr><td>cab|cat
   ^</td><td>4</td><td>c
0</td><td>a
0</td><td>t
1</td><td>0
0</td><td>0
0</td><td>0
0</td><td>0
0</td><td>0
0</td><td>Alternation char does not shift in.</td></tr>
<tr><td>cab|cat
  ^</td><td>5</td><td>b
1</td><td>c
0</td><td>a
0</td><td>t
1</td><td>0
0</td><td>0
0</td><td>0
0</td><td>0
0</td><td>
End of another subpattern.</td></tr>
<tr><td>cab|cat
 ^</td><td>6</td><td>a
0</td><td>b
1</td><td>c
0</td><td>a
0</td><td>t
1</td><td>0
0</td><td>0
0</td><td>0
0</td><td></td></tr>
<tr><td>cab|cat
^</td><td>7</td><td>c
0</td><td>a
0</td><td>b
1</td><td>c
0</td><td>a
0</td><td>t
1</td><td>0
0</td><td>0
0</td><td></td></tr>
</table>

This trace yields precisely the same values shown in Figure 3.12. Determining the length of *P* will require one pass through the entire pattern. Then the *while-loop* of the algorithm examines every character of the pattern exactly once and generates a cell entry for every pattern character except the alternation operator which shares the *EOP Latch* with the last character of every subpattern. In the "cab|cat" example, the pattern was seven characters long. This resulted in seven iterations of the *while-loop*, producing six cells for the pattern. In the general case, any pattern of length *m* will require at most *2m* steps to preprocess and *m-number_of_alternations* cells. The algorithm, therefore, has a linear complexity denoted as *O(m)*.

While uniprocessor software algorithms may also claim linear preprocessing, the coefficient of *m* can be quite large and the hardware algorithm can still show significant improvement despite common complexity levels. However, the software algorithms lose linearity with the introduction of wild card characters. As shown in the following section, the hardware algorithm easily addresses wild cards with no impact on complexity.

3.2.3. Wild Cards

Wild cards are not generally considered regular expression operators and are seldom encountered in language theory. They can, however, be defined through regular expression operators and are an extremely useful and common shorthand in defining a search pattern. Two types of wild cards are defined and designed into the hardware algorithm. They are "Fixed Length Don't Care" and "Variable Length Don't Care" characters.

3.2.3.1. Fixed Length Don't Care

The "Fixed Length Don't Care" character *(fldc)* will be denoted by a period (.) and it will state that its position in the pattern can match any single element of the alphabet. For example, let $\Sigma = \{a, b, c\}$ and the pattern P = *"a.c"*. Then P represents a finite set that consists of the patterns *"aac"*, *"abc"*, and *"acc"*. Any number of *fldc* characters can appear at any position in the pattern.

This can be defined in regular expression terms as follows:

(1) Let Σ be the set representing the alphabet.

(2) Let the number of elements in Σ be represented by $|\Sigma|$.

(3) Let α be a single element of Σ.

(4) Let α_i be the i^{th} element of Σ (i.e., α_1 = the first element of Σ and $\alpha_{|\Sigma|}$ = the last element of Σ).

(5) The fldc character is the equivalent to

$$\alpha_1 \mid \alpha_2 \mid \bullet\bullet\bullet \mid \alpha_{|\Sigma|}$$

This definition in effect says that δ of the FSA five-tuple will have a transition to the next state for every element of Σ. The hardware algorithm, as currently defined for multiple exact patterns, maps δ transitions directly to the *Pattern Latch* values and has not had the problem of multiple transitions associated with a given state. One solution might be to have |Σ| *Pattern Latches* and *Comparators* in each cell with the outputs *or-ed* together. This would always yield a 1 since one of the *Pattern Latches* would always match whatever character was present. Instead, a more efficient solution will use a single latch to represent this special case as is shown in the following diagram.

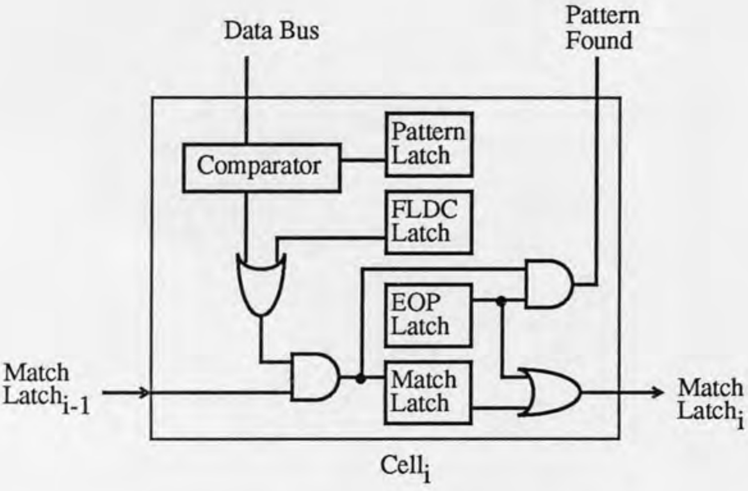


Figure 3.13. FLDC Hardware Addition

If the *FLDC Latch* is set to 0, then once again the logic of the cell is unchanged from the previous version. If it is set to 1, then the *Comparator Logic* is overridden and a logic 1 is continuously presented to the *and-gate* input. Because the *and-gate* still requires the *Match Latch*$_{i-1}$ input to be a 1 before it can set the *Match Latch*, concatenation and proper sequencing are maintained.

The introduction of the *fldc* character has added some new rules to the construction of a searcher. First, when encountered, a transition on all elements of Σ is used instead of a transition on a single character when constructing the NFSA. Secondly, such a transition in the NFSA is implemented in hardware by setting the *FLDC Latch* to 1. Figure 3.14 shows the new recursive rule for the NFSA construction of a regular expression that can contain an *fldc*. Also contained in

Figure 3.14 is an instantiation of the hardware algorithm which alters the "cab|cat" example to be "c.b|c.t". The *Pattern Latch, EOP Latch,* and *FLDC Latch* values are displayed respectively from top to bottom in each cell.

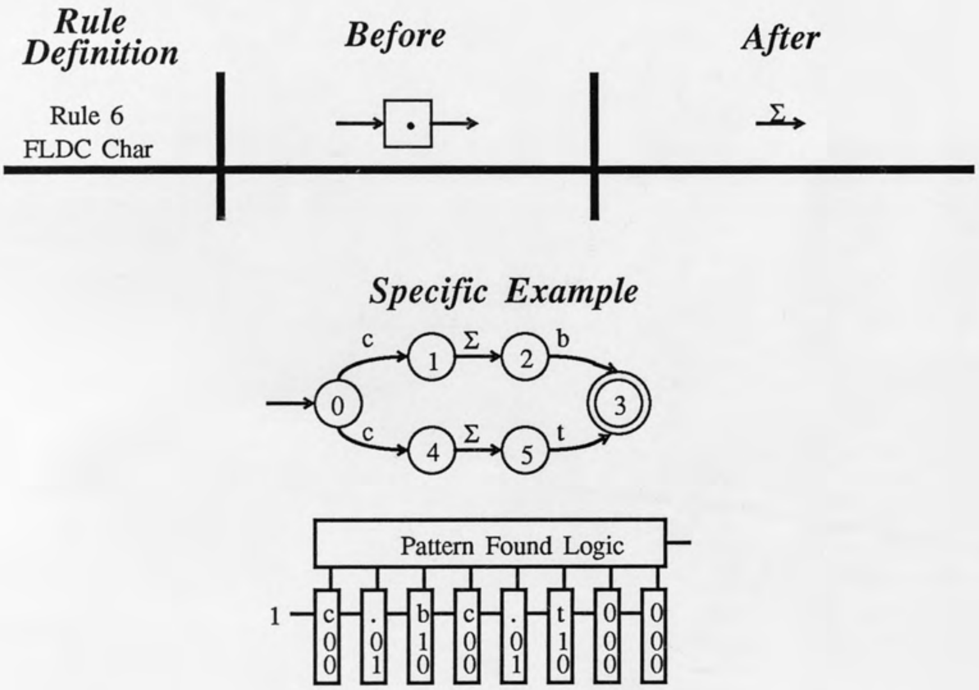


Figure 3.14. Hardware for 'c.b|c.t'

If the string "ccab" was applied to these examples, then the following would occur:

TABLE 3.6. Trace of 'ccab'

Input Character	NFSA States Active	*Match Latch*$_i$ with value = 1
c	1,4	1,4
c	1,2,4,5	1,2,4,5
a	2,5	2,5
b	3	3

The preprocessing algorithm to prepare the hardware for searching *fldc* characters requires setting the *FLDC Latch* to a 1 when an *fldc* is encountered. The preprocessing algorithm will be explicitly defined once again after the *Variable Length Don't Care* character is discussed.

3.2.3.2. Variable Length Don't Care

The "Variable Length Don't Care" *(vldc)* will be denoted by a question mark (?) and it will state that its position in the pattern can match any number of elements from the alphabet. Using the example $\Sigma = \{a, b, c\}$ and P = *"a?c"*, then P represents an infinite set of patterns consisting of *"aac", "abc", "acc", "aaac", "aabc", "aacc", "abac",* and so on, more simply stated as any string of length three or greater that starts with "a" and ends with "c". As with the *fldc* character, a *vldc* character can appear any number of times at any position in the pattern.

Since the *fldc* character (denoted by a period) has already been defined and implemented in the hardware algorithm, the *vldc* character could be defined through two consecutive *fldc* characters with a closure operator on the second ("..*"). This would match any single character followed by zero or more of any other characters. Using only traditional regular expression operators combined with the notation of the previous section, this is equivalent to:

$$(\alpha_1 \mid \alpha_2 \mid \bullet\bullet\bullet \mid \alpha_{|\Sigma|})(\alpha_1 \mid \alpha_2 \mid \bullet\bullet\bullet \mid \alpha_{|\Sigma|})^*$$

Applying the rules for NFSA construction to this subpattern ("..*") would cause the sequence of steps illustrated in Figure 3.15. Defining the token question mark (?) to represent ("..*") allows the new rule 7 in Figure 3.16 to be defined.

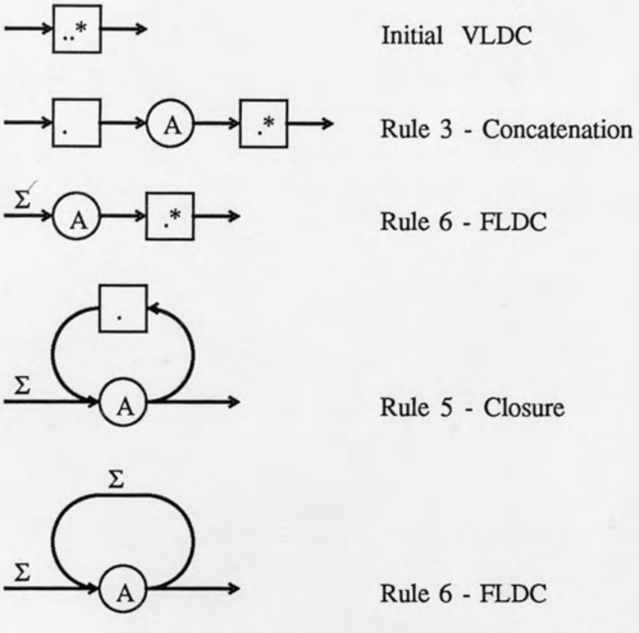


Figure 3.15. '..*' NFSA Construction

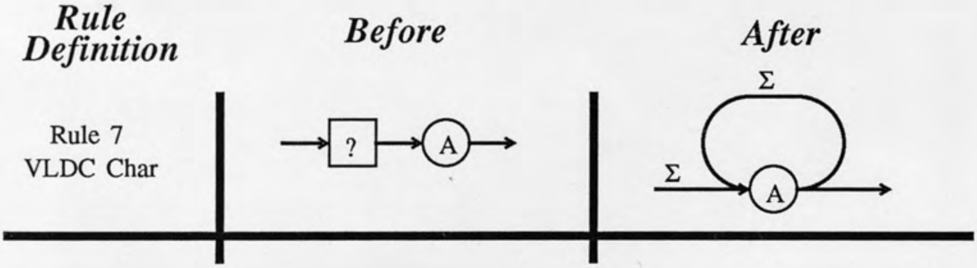


Figure 3.16. VLDC NFSA Construction Rule

This is implemented in the hardware cell by the addition of another latch and some combinational logic as shown in the following diagram.

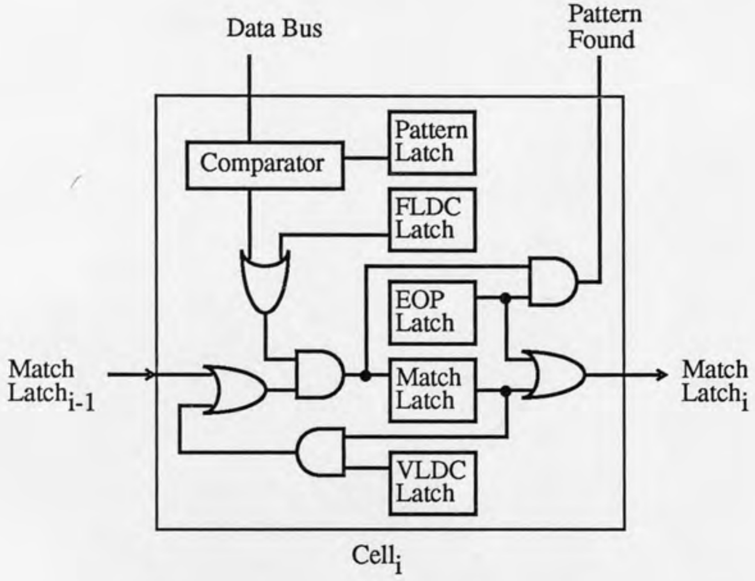


Figure 3.17. VLDC Hardware Addition

Setting the *VLDC Latch* to 0 leaves the cell unchanged from the previous definition. A 1 in both the *FLDC Latch* and *VLDC Latch* implements the functionality of the *vldc* character. The *FLDC Latch* set to 1 implements the Σ transition into the state. The feedback logic of the *VLDC Latch* implements the Σ transition out of and back to the state. Once *Match Latch*$_{i-1}$ becomes a 1 for the first time, *Cell*$_i$ will become an active state providing a 1 output on *Match Latch*$_i$ until cleared for a new search.

Modifying the "cab|cat" example to be "c?b|c.t" would yield the NFSA and hardware implementation as shown in Figure 3.18. The hardware cell now contains the *Pattern Latch, EOP Latch, FLDC Latch,* and *VLDC Latch,* the contents of which are displayed from top to bottom in each cell.

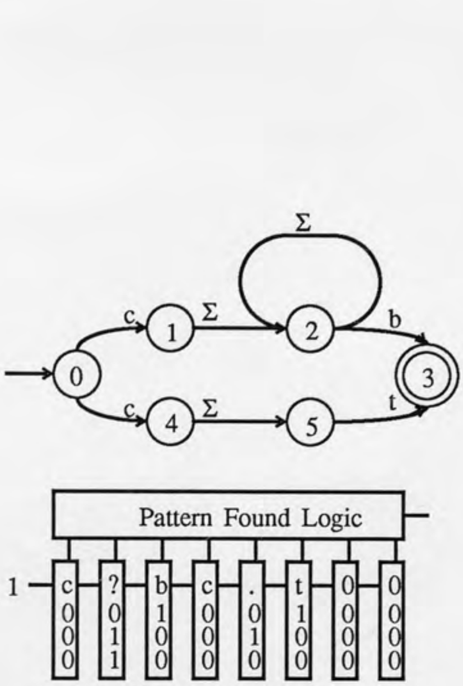


Figure 3.18. ‘c?b|c.t’ NFSA/Hardware Example

Tracing the input string "cctbb" yields some interesting results for this particular example.

TABLE 3.7. Trace of ‘cctbb’

Input Character	NFSA States Active	*Match Latch*$_i$ with value = 1
c	1,4	1,4
c	1,2,4,5	1,2,4,5
t	2,3,5	2,3,5
b	2,3	2,3
b	2,3	2,3

We note that the first three characters of the string ("cct") lead to state 3 and a match of the "c.t" subpattern. Those same three characters are also a prefix to matching "cctb" and "cctbb" to the "c?b" subpattern. Once state 2 of this example becomes active, it will remain active for the remainder of the search. The transition on a "b" input character non-deterministically makes both states 2 and 3 active simultaneously.

The introduction of additional non-determinism to the NFSA implies additional overhead for the uniprocessor, software algorithms. The only impact to the hardware algorithm was some feedback logic internal to the cell. The preprocessing algorithm to load the hardware is now:

```
pat_ptr = length(P);
alternation_flag = 1;
while (pat_ptr > 0) do
begin
  Pattern = EOP = FLDC = VLDC = 0;
  switch (P[pat_ptr])
  begin
        case '|':
          alternation_flag = 1;
          break;
        default:
          Pattern = P[pat_ptr];
          switch (P{pat_ptr)
          begin
                case '?': VLDC = 1;
                case '.': FLDC = 1;
          end;
          if (alternation_flag) then do
          begin
                EOP = 1;
                alternation_flag = 0;
          end;
          shift_pattern_in();
  end;
  pat_ptr = pat_ptr - 1;
end;
```

The only changes to the algorithm are the initialization of two more latch inputs and the switch statement to set the appropriate latches if a *don't care* character is encountered. For an arbitrary pattern of length *m*, the algorithm maintains at most *2m* examinations and still requires only *m - number_of_alternations* cells. Therefore, the algorithm remains *O(m)* in complexity.

3.2.4. Regular Expression Operators

The section *Concepts in Searching* defined five rules for the generation of an NFSA from an arbitrary regular expression. Two more "searching algorithm specific" rules were introduced to address the frequently used *don't care* characters. These seven rules are shown together here.

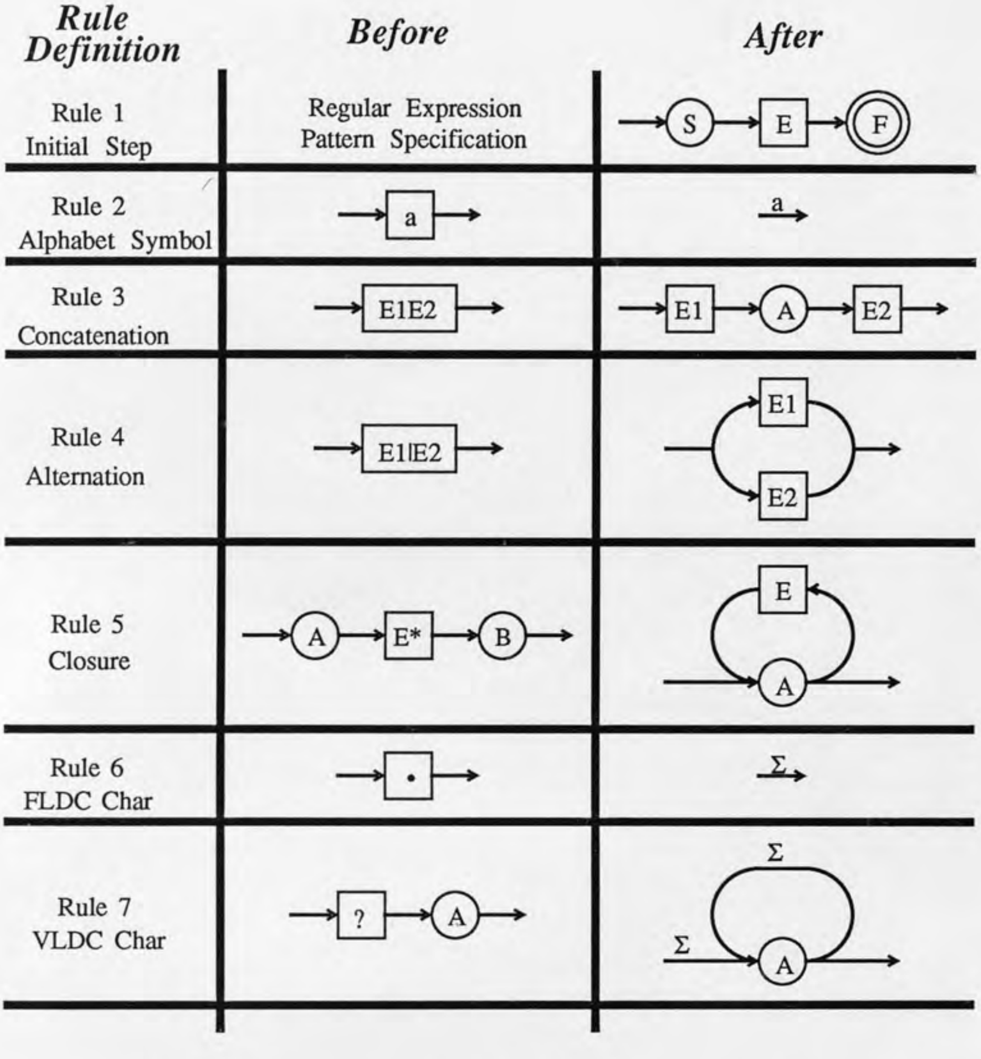


Figure 3.19. NFSA Construction Rules for Regular Expressions with Wild Cards

Up to this point, the algorithm has restricted the pattern specification from using parentheses and closure. With those restrictions, linearity in both preprocessing and hardware cell requirements has been maintained. Furthermore, the *EOP Latch* allowed all of the cells to be connected in a one-dimensional array using only adjacent cell communication. Allowing parentheses introduces the complications that the output of one state may be required to be the input to several other states, or a single state may have multiple inputs. Introducing closure complicates routing matters even further.

Consider the on-going "cab|cat" example. An equivalent regular expression for the same pattern could be "ca(b|t)". Likewise, consider the regular expression "a(b|c)*d" utilizing both parentheses and closure. The state diagrams for the NFSAs that would be generated by these expressions are shown here.

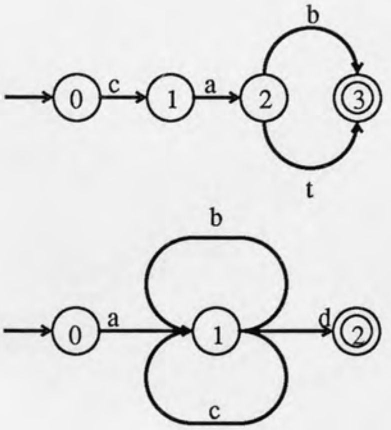


Figure 3.20. NFSA State Diagrams for 'ca(b|t)' and 'a(b|c)*d'

In the "ca(b|t)" example, conversion to the hardware cell forces the problem of two different characters being able to cause the transition from state 2 to 3. State 1 of the "a(b|c)*d" example has three input transitions (on an "a", "b", or "c") and three output transitions (on a "b", "c", or "d"). Given that the *Pattern Latch* of a cell can only hold one element from Σ, the first step towards resolving this dilemma is requiring a cell for each arc of the NFSA. Our examples could then be laid out in hardware cells as follows.

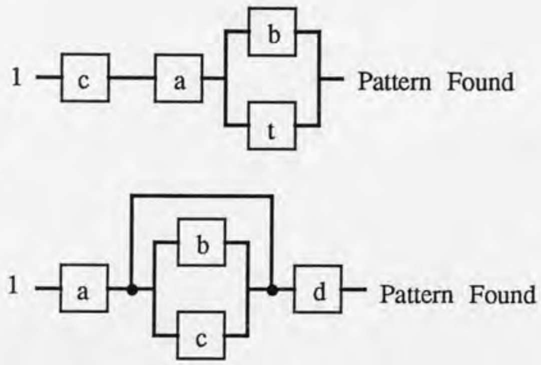


Figure 3.21. Routing for 'ca(b|t)' and 'a(b|c)*d'

Unfortunately, this has once again caused the algorithm to be inflexible due to the custom routing of inter-cell communication lines. The *EOP Latch* was able to resolve this problem when each subpattern was guaranteed to be the boundary of a search pattern and communications remained limited to adjacent cells. Now, each subpattern might simply be some small portion of an individual search pattern and may need to pass its state information to several other cells.

The custom routing problem can be simplified by placing the cells in a horizontal array and then routing the cell *Match Latch*$_{i-1}$ inputs and *Match Latch*$_i$ outputs to specific signal lines arranged as a bus across the width of cells. If a means were made available to reprogram a cell to read the value of one of those signal lines or write a value to one of those signal lines, then no custom hardwiring would be required at all. Consider replacing the *EOP Latch* with two other latches labeled *Read Latch* and *Write Latch.* If both *Read Latch* and *Write Latch* are zero, then the cell would function as defined earlier. If either or both of these latches are set to 1, then the cell takes on an entirely new definition. Instead of holding the character of a state transition in the *Pattern Latch,* a number representing a bus signal line is placed in the *Pattern Latch.* A 1 in the *Read Latch* indicates to read the value on the bus signal line denoted by the contents of the *Pattern Latch* and place that value on the *Match Latch*$_i$ output of this cell. A 1 in the *Write Latch* indicates to write the value of the *Match Latch*$_{i-1}$ of this cell to the bus signal line denoted by the

Pattern Latch. Both latches can be set to 1 allowing *Match Latch*$_{i-1}$ to pass through to *Match Latch*$_i$ as well as setting the bus line.

The following figure takes the "a(b|c)*d" example from an NFSA state diagram to a custom routed cell layout to a conceptual bus interconnection to an actual hardware cell instantiation with the *Pattern Latch, Read Latch, Write Latch, FLDC Latch,* and *VLDC Latch* displayed from top to bottom.

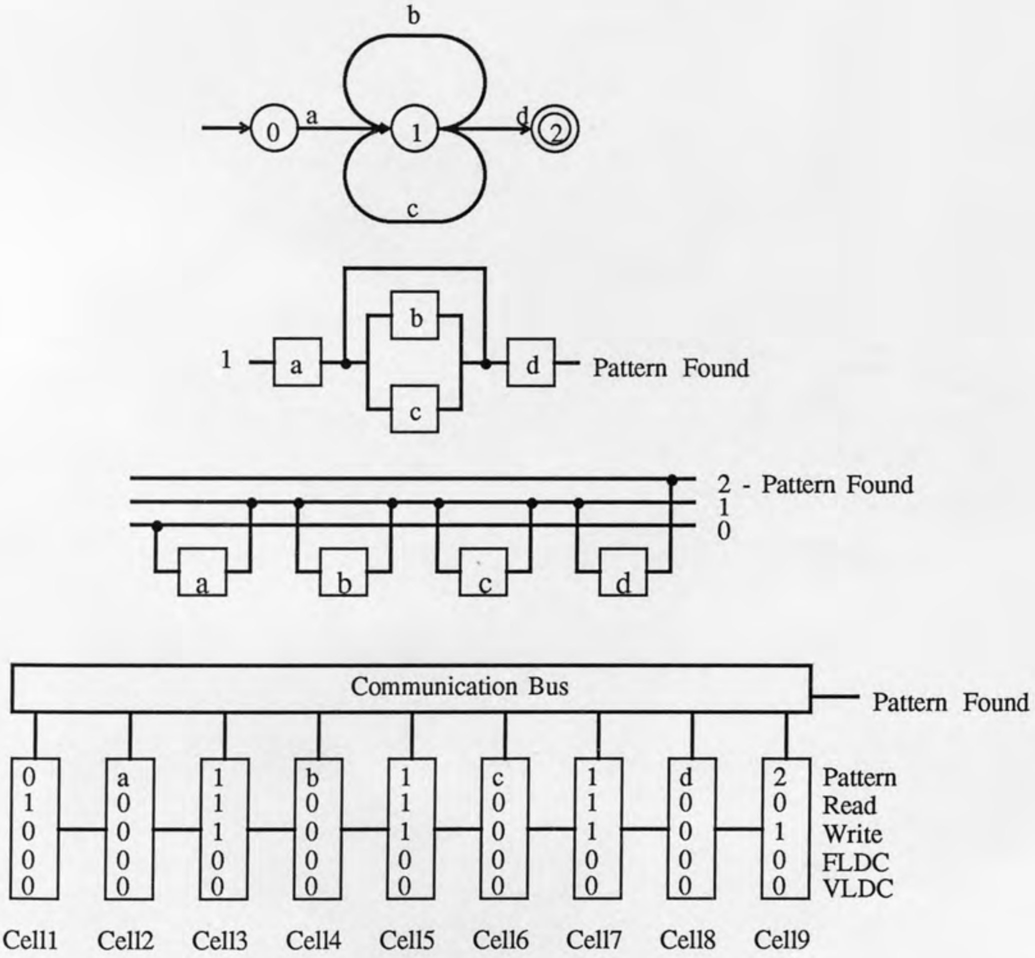


Figure 3.22. Implementations of 'a(b|c)*d'

This example serves to illustrate the cell used as both a match-cell and routing-cell. The even cells (2, 4, 6, and 8) use the definition already established from the previous sections (direct assignment of arc transitions to *Pattern Latch* values). The odd cells contain the communication bus assignments which define the input and output lines to the even cells. In this particular example, the state number maps directly into the *Pattern Latch.*

The hardware to accomplish this new definition of the cell is shown in the following diagram. The logic for one of the bus lines is also shown.

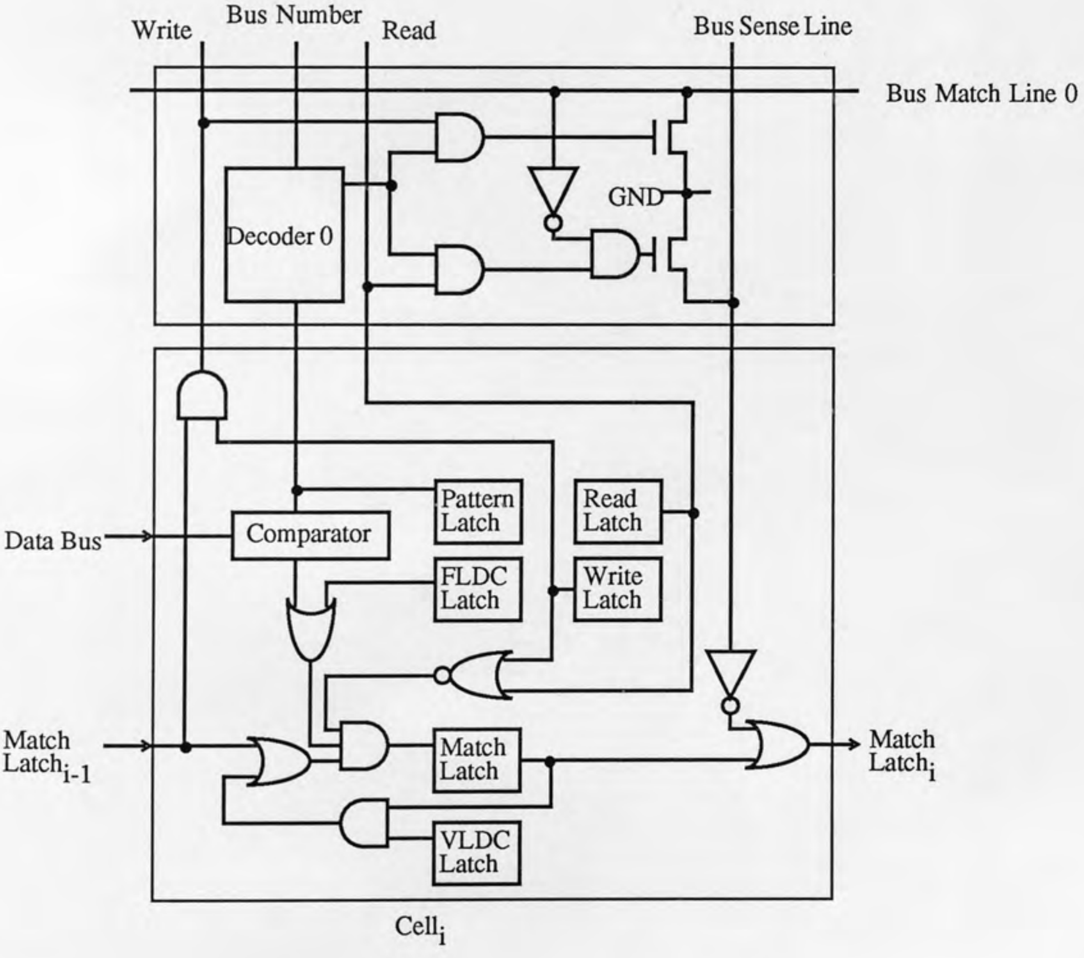


Figure 3.23. Regular Expression Hardware Cell

The values of the *Read Latch* and *Write Latch* are *nor-ed* together to disable the normal *Match Latch* logic if the cell definition is to be used as a routing cell. Their values are also passed up through the bus logic to appropriately read and/or write a bus line.

The bus logic of Figure 3.23 contains some logic that is somewhat different from normal combinational logic. The diagram makes use of *pass transistor logic* as would be found in field effect transistors and Metal Oxide Semiconductor (MOS) technology. The pass transistor functions as a switch which will only pass a signal through if its gate is a 1. Hence, if the *Write Latch* is set to 1 and *Match Latch*$_{i-1}$ is 1 and the decoder yields a 1 from the *Pattern Latch*$_i$ value, then a 1 is placed on the gate of the pass transistor connecting the *Bus Match Line* to ground, thus setting the *Bus Match Line* to 0. Both the *Bus Sense Line* and *Bus Match Line* are normally pulled high unless grounded. These lines are logically active low.

While the same result could have been accomplished using gate logic, the pass transistor was chosen for several reasons. The amount of hardware and the repetitive nature of the bus and cells lends the algorithm quite well towards implementation in MOS technology. If combinational logic were used on the *Bus Match Line*, then a cumulative gate delay would be incurred through each cell connection to the bus causing an excessive critical path. Likewise, the *Bus Sense Line* would have an excessive gate delay vertically through the bus if gate logic were applied to it. Instead, both of these lines can be implemented as an *inverted-wired-or* which is active low. This allows the *Bus Match Line* to stay high normally but can be pulled low by any number of cells simultaneously. The *Bus Sense Line* will stay high unless a read is initiated and the respective *Bus Match Line* was low. The decoder can also be easily implemented in MOS using pass transistors as shown here for a two-bit decoder.

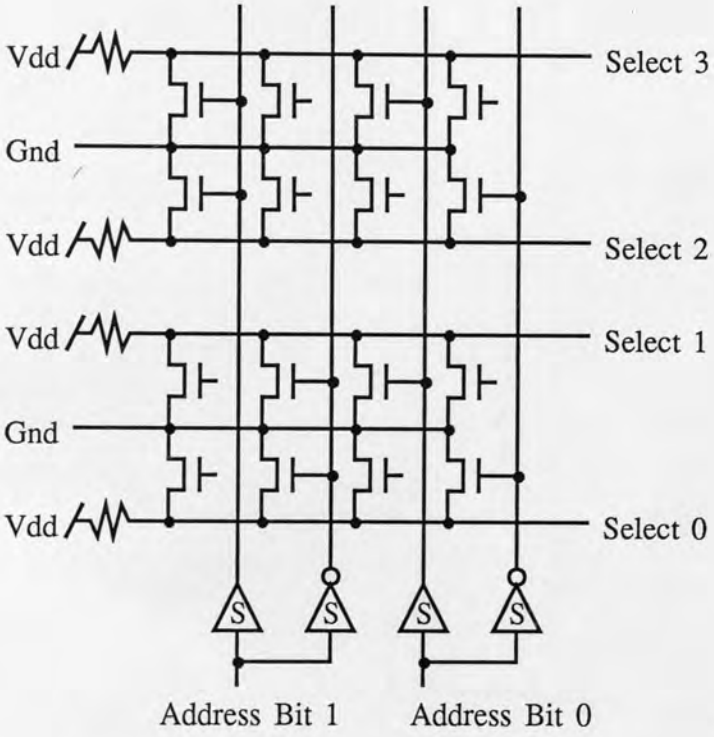


Figure 3.24. Two-Bit Decoder Using Pass Transistors

Each of the *Select* lines is normally pulled high by the resistor connected to Vdd. The address bits and their complements are amplified and broadcast vertically through the decoder. Gates to pass transistors are then connected to the appropriate address lines composing the address for selection. The selection output of the decoder can be inverted for *active-high* logic or left untouched for *active-low* logic.

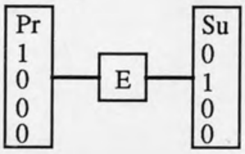
3.2.4.1. Converting a Full Regular Expression to Its Hardware Instantiation

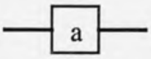
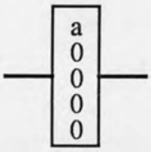
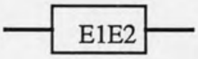
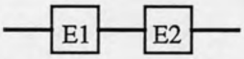
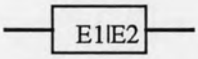
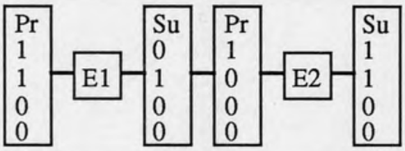
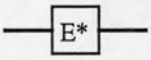
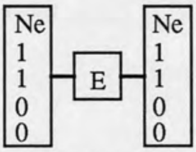
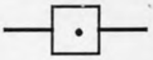
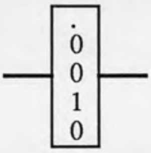
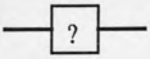
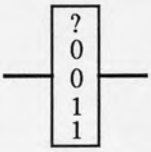
Since the purpose of this algorithm is searching, we still must address the special connections for the start of the pattern and a successful match. To that end, the *Bus Match Line*$_0$ is now defined and reserved as the *Pattern Found* line and the *Bus Match Line*$_1$ is defined and reserved for the start of a pattern. *Bus Match Line*$_1$ will always be active since any character can potentially be the start of another matched pattern. The remainder of the communication bus lines only serve to connect cell inputs and outputs with no special meaning.

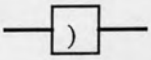
Rather than converting the final NFSA generated by the rules in Figure 3.17 to its hardware equivalent, it is easier to define the conversion while the rules are being applied as shown in Figure 3.25.

Some minor differences have been introduced in these rules. The variables *Prefix, Suffix,* and *Next* have been introduced to properly handle assignments to the communication bus. *Prefix* and *Suffix* define incoming and outgoing state information, respectively, and are initialized to 1 and 0 to coincide with the *Bus Match Line*$_1$ and *Bus Match Line*$_0$ definitions. Each hardware cell is shown with the *Pattern Latch, Read Latch, Write Latch, FLDC Latch,* and *VLDC Latch* shown from top to bottom. If *Pr, Su,* or *Ne* appears in a *Pattern Latch,* then the *Prefix, Suffix,* or *Next* value is placed there. The regular expression is stepwise refined through these rules from right to left. This is done so that the final preprocessing algorithm can find the unary closure operator before the expression on which it is operating. In constructing the NFSA state diagrams, parentheses were not directly considered since multiple arcs could be placed from one state to another. Each parenthesis token is individually addressed in these rules to explicitly define which *Bus Match Line* to read and/or write. A stack and the *Next* variable are employed to address nesting. As deeper levels are entered, the *Next* variable supplies an unused bus line. As levels are returning upward, the stack restores the prefix and suffix values for the appropriate level.

Rule Definition	***Before***	***After***

Rule 1
Initial Step
	Regular Expression
	Pattern Specification
		Suffix = 0;
		Prefix = 1;
		Next = 2;
		

Rule 2 Alphabet Symbol		
Rule 3 Concatenation		
Rule 4 Alternation		 Pr 1 1 0 0 E1 Su 0 1 0 0 Pr 1 0 0 0 E2 Su 1 1 0 0
Rule 5 Closure		 Next = Next + 1;
Rule 6 FLDC Char		
Rule 7 VLDC Char		

Rule 8
Close Parenthesis
		Push Prefix and Suffix on Stack;
		Suffix = Next; Prefix = Next + 1;
		Next = Next + 2;

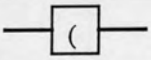
Rule 9 Open Parenthesis		Pop Prefix and Suffix from Stack;

Figure 3.25. Rules for Loading the Hardware Cells

Applying these rules to the "a(b|c)*d" example would yield the following hardware implementation.

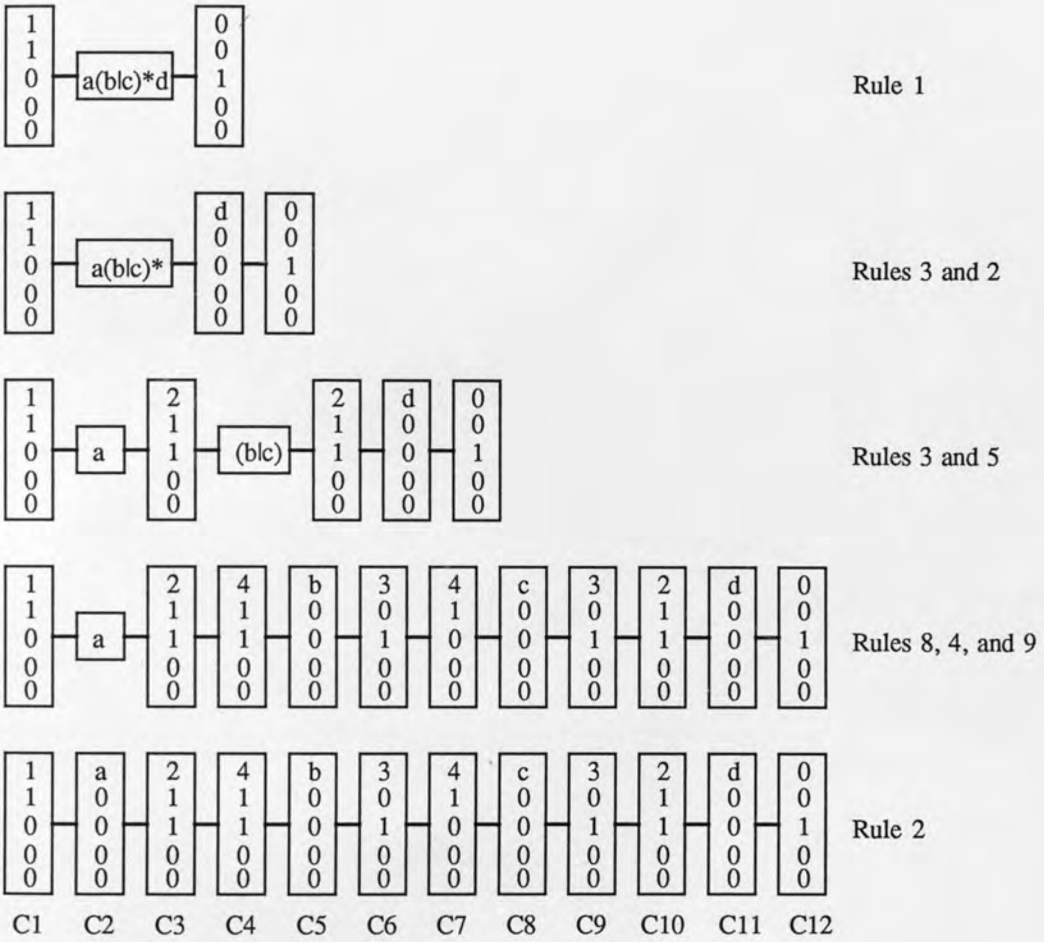


Figure 3.26. Application of Rules for Loading the Hardware Cells

The final count of hardware cells in this figure is greater than the count of hardware cells in the implementation of Figure 3.22, but this still finds the same patterns. Furthermore, the bus is larger as implied by the reference to *Bus Match Line*$_4$ in cells C4 and C7. Additional heuristics that take advantage of knowing the scope of the closure operator can reduce the cell count and bus size closer to their optimum values. Meanwhile, this figure illustrates that bus-routing cells **can** be adjacent and in some cases, **must** be adjacent. To illustrate the routing mechanisms employed here, a trace of the input string "adacbd" is shown in this table.

TABLE 3.8. Trace of 'adacbd'

Input Character	Active Cells	Comments
a	1,2,3,4,7,10	Initial State - Cell1 always active Cell1 active allows Cell2 to match input Cell2 activates Cell3 Cell3 activates Cell4 and Cell10 Cell4 activates Cell7
d	1,11,12	Cell2 deactivates Cell3 Cell3 deactivates Cell4 and Cell10 Cell4 deactivates Cell7 Cell10 having been active allowed Cell11 to activate Cell11 activates Cell12 - Pattern Found
a	1,2,3,4,7,10	
c	1,3,4,7,8,9,10	Cell8 activates Cell9 Cell9 activates Cell10 etc.
b	1,3,4,5,6,7,10	
d	1,11,12	Pattern Found

Because each *Bus Match Line* is a **bus** line, communication is broadcast to all cells in both directions. Hence, Cell2 was able to activate Cell10 and vice-versa. When any particular *Bus Match Line* is written by one cell, the information is propagated to all other cells that read from that *Bus Match Line*.

3.2.4.2. Complexity Analysis

Having a technique to implement regular expressions with wild cards, we now must quantify how many cells will be required for an arbitrary expression as well as how many bus lines. An approximation of these quantities can be determined through examinationn of the NFSA constructed from an arbitrary regular expression. Floyd and Ullman (1980) prove that a maximum of *4m* arcs and *2m* states will be created. Mapping every arc of the resulting NFSA to a match cell and surrounding each of those match cells with routing cells containing the prefix state and suffix stat information results in a hardware searcher for the regular expression. This gives an upper bound on the algorithm of *4m* cells for the arcs plus *2(4m)* cells for the states surrounding each arc plus *2m* bus lines for those states. This implies a length of *12m* cells and height of *2m* bus lines for *24m*^{2} area. However, these complexities are not strictly accurate since simple concatenation does not require use of the *Bus Match Lines* and, as already demonstrated, heuristics can be applied to achieve smaller cell counts and bus sizes. A preprocessing algorithm that employs some of those heuristics is given in Appendix 2 written in the C (Kernighan and Ritchie 1978) programming language for any reader wishing to implement the program. A run of the preprocessing program on the "a(b|c)*d" example would yield the following output.

TABLE 3.9. Output of Preprocessing for 'a(b|c)*d'

```
input pattern  =  a(b|c)*d
pattern        =    1    a  2  b  2  c  2  d  0
read           =    1    0  1  0  1  0  1  0  0
write          =    0    0  1  0  1  0  1  0  1
fldc           =    0    0  0  0  0  0  0  0  0
vldc           =    0    0  0  0  0  0  0  0  0
```

This is better than the result from the strict application of the recursive rules as shown in Figure 3.26 and equivalent to the result shown in Figure 3.22. As in previous versions of the preprocessing algorithm, a pointer moves from the end of the pattern to the beginning examining and processing each character one at a time. The heuristics are applied through the use of a variable called *closure_flag*. If the closure is associated with a parenthesized subexpression, then the close parenthesis ")" can set the prefix and suffix to the same value and collapse the redundant adjacent cells. This is best illustrated by the differences in the results of Figures 3.22 and 3.26 shown again here.

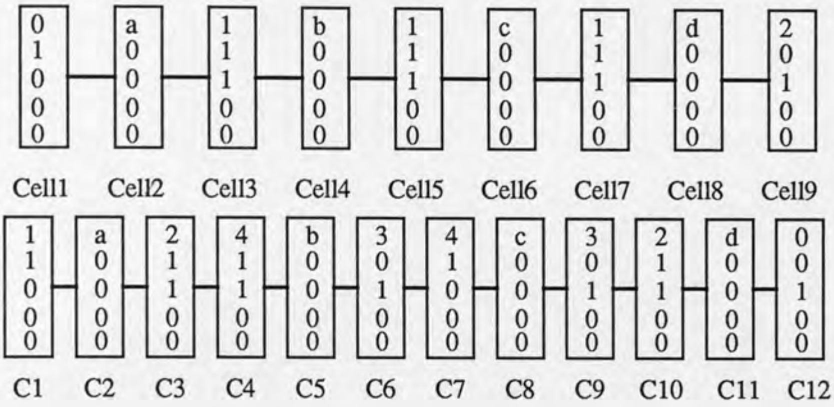


Figure 3.27. Two Hardware Implementations of 'a(b|c)*d'

Because the "(b|c)" subexpression is under closure, the "b", "c", and "d" can all share the same prefix and the "a", "b", and "c" can all share the same suffix. If one cell is *writing* a suffix and the next cell is *reading* a prefix identical to the suffix written in the previous cell, then the two cells can collapse to one, setting both the *Read Latch* and *Write Latch* to 1.

A *bus_num* function in the preprocessing algorithm uses the *closure_flag* variable to determine how to set the suffix and prefix. Combined with the knowledge of the parentheses nesting level and the *closure_flag*, *bus_num* manages the prefix and suffix variables and returns the appropriate value.

The closure heuristic brings the preprocessing algorithm close to optimum cell assignment but fails in some cases. If unnecessary parentheses are used, the algorithm does not recognize this and generates unnecessary routing. A trivial example is the regular expression "(a|b)". The cell assignment from the algorithm and the more optimal assignments are shown here.

TABLE 3.10. Preprocessing of Unnecessary Parentheses

input pattern	=	(a|b)							
pattern	=	1	3	a	2	3	b	2	0
read	=	1	1	0	0	1	0	1	0
write	=	1	1	0	1	0	0	1	1
fldc	=	0	0	0	0	0	0	0	0
vldc	=	0	0	0	0	0	0	0	0

input pattern	=	(a|b)					
pattern	=	1	a	0	1	b	0
read	=	1	0	0	1	0	0
write	=	1	0	1	0	0	1
fldc	=	0	0	0	0	0	0
vldc	=	0	0	0	0	0	0

Furthermore, if parenthesized subexpressions are immediately adjacent with no intervening concatenation, then the suffix of the first subexpression could be used as the prefix of the next. The preprocessing algorithm does not catch this either as demonstrated by the pattern "(a|b)(c|d)".

TABLE 3.11. Preprocessing of Adjacent Parenthesized Subexpressions

input pattern	=	(a|b)(c|d)													
pattern	=	1	5	a	4	5	b	4	3	c	2	3	d	2	0
read	=	1	1	0	0	1	0	1	1	0	0	1	0	1	0
write	=	0	1	0	1	0	0	1	1	0	1	0	0	1	1
fldc	=	0	0	0	0	0	0	0	0	0	0	0	0	0	0
vldc	=	0	0	0	0	0	0	0	0	0	0	0	0	0	0

input pattern	=	(a|b)(c|d)										
pattern	=	1	a	2	1	b	2	c	0	2	d	0
read	=	1	0	0	1	0	1	0	0	1	0	0
write	=	0	0	1	0	0	1	0	1	0	0	1
fldc	=	0	0	0	0	0	0	0	0	0	0	0
vldc	=	0	0	0	0	0	0	0	0	0	0	0

Despite these suboptimal intricacies, the preprocessing algorithm can provide a quantification of the complexity requirements of the searching algorithm. If the original regular expression had *m* characters in it (inclusive of the operators and wild cards), then the number of cells required is defined as follows:

(1) Let m be the number of tokens in the original regular expression.

(2) Let a be the number of alternation operators which are not within the scope of a closure.

(3) Let b be the number of single characters in closure, i.e., $b = \{\alpha^* \mid \alpha \in \Sigma\}$.

(4) Let c be the number of close parentheses, closure pairs ")*".

(5) The total number of cells which would be required is then defined by the equation:

$$Length = m + 2 + a + b - c$$

The constant 2 is the result of the immediate assignment of *Bus Match* $Line_1$ and *Bus Match* $Line_0$ for the start state and final state. Every alternation that is not in closure (a in the equation) gets turned into a suffix-write cell followed by a prefix-read cell and therefore adds 1 to the overall length. An alternation in closure becomes a cell which both reads and writes the same prefix/suffix value and therefore neither adds nor subtracts from the length. A single element in closure (b in the equation) has a prefix-read-write cell and suffix-read-write cell placed around it adding 1 to the overall length. A close-parenthesis, closure pair (c in the equation) maps into a single suffix-read-write cell and therefore reduces the overall length by 1.

The height of the bus is defined by the following:

(1) Let b and c be defined as before.

(2) Let d be the number of close parenthesis ")" not adjacent to a closure operator.

(3) The number of bus lines required is then defined by the equation:

$$Height = 2 + 2\,d + c + b$$

Once again, the constant 2 represents the start state and final state assignments of *Bus Match Line* 1 and 0. Every parenthesized expression which is **not** in closure (d in the equation), adds both a new prefix and a new suffix. Every subexpression which **is** in closure (c and b in the equation), adds the same prefix and suffix.

The variables a, b, c, and d are all dependent on m. The maximum value a can obtain in a valid regular expression is *floor(m/2)* (i.e., the integer value, rounded down, of m divided by 2). For example, the pattern P = "*a|b|c|d*" has $m=7$ and $a=3$. The variable b has a maximum of $m/2$

since *b* is built from two characters. The number of close parentheses also has a maximum of *m/2* since they always must be matched with an open parenthesis. The number should be much less than *m/2* since parentheses are unnecessary unless surrounding subexpressions.

Some extreme examples which attempt to exercise these calculations are shown here.

TABLE 3.12. Examples of Length and Height Calculations

```
input pattern  =  (abc|def)(ghi|jk*l|mn(op|qr)*st)uv
Length         =  34 + 2 + 3 + 1 - 1 = 39
Height         =   2 + 4 + 1 + 1 = 8
pattern = 1 7 a b c 6 7 d e f 6 3 g h i 2 3 j 5 k 5 l 2 3 m n 4 o p 4 q r 4 s t 2 u v 0
read    = 1 1 0 0 0 0 1 0 0 0 1 1 0 0 0 0 1 0 1 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0 0
write   = 0 1 0 0 0 1 0 0 0 0 1 1 0 0 0 1 0 0 1 0 1 0 1 0 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1
fldc    = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
vldc    = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

      input pattern  =  a*bc(de*f|ghi*|j*(kl|m)*)*nop
      Length         =  29 + 2 + 0 + 4 - 2 = 33
      Height         =   2 + 0 + 2 + 4 = 8
      pattern        = 1 7 a 7 b c 2 d 6 e 6 f 2 g h 5 i 5 2 4 j 4 3 k l 3 m 3 2 n o p 0
      read           = 1 1 0 1 0 0 1 0 1 0 1 0 1 0 0 1 0 1 1 1 0 1 1 0 0 1 0 1 1 0 0 0 0
      write          = 0 1 0 1 0 0 1 0 1 0 1 0 1 0 0 1 0 1 1 1 0 1 1 0 0 1 0 1 1 0 0 0 1
      fldc           = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
      vldc           = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Note that the suboptimal nature of the preprocessing algorithm is demonstrated in the first example. The "a" and "d" could have been preceded by *Bus Match Line*$_1$ rather than *Bus Match Line*$_7$ and the "c" and "f" could have been succeeded by *Bus Match Line*$_3$ eliminating *Bus Match Line*$_7$ and *Bus Match Line*$_6$ completely. However, the length and height equations correctly calculate their values for the algorithm as it stands.

These two equations provide the means to determine worst case behavior in the algorithm. The constant 2 is unaffected by *m* (the length of the regular expression). The number of alternations not in closure adds to the length but not the height. Every close-parenthesis, closure pair adds to the height but subtracts from the length. The close-parenthesis not in closure adds two to the height but requires an open parenthesis somewhere else in the expression and the length is unaffected. The one common factor that adds to both length and height and requires no other operators to appear in the expressions is the single element in closure (α^*). If a regular expression of length m were composed of a series of single elements all with closure operators, then the total number of cells required by the algorithm and the height of the bus would be:

$$Length = 2 + m + 0.5m = 2 + 1.5m$$

$$Height = 2 + 0.5m$$

The $0.5m$ is derived from the fact that it takes two elements from the regular expression (the element plus the closure operator) to add one to the length and height. A simple example is the pattern $P = "a^*b^*c^*"$.

TABLE 3.13. Example of Worst Case Expansion

input pattern	=	a*b*c*										
Length	=	2 + 1.5(6) = 11										
Height	=	2 + 0.5(6) = 5										
pattern	=	1	4	a	4	3	b	3	2	c	2	0
read	=	1	1	0	1	1	0	1	1	0	1	0
write	=	0	1	0	1	1	0	1	1	0	1	1
fldc	=	0	0	0	0	0	0	0	0	0	0	0
vldc	=	0	0	0	0	0	0	0	0	0	0	0

We now have a quantification of the hardware requirements for the regular expression hardware searching algorithm. The preprocessing algorithm first determines the length of the expression and then processes one element at a time. This is still linear $O(m)$ as with all the previous preprocessing algorithms. However, as it processed each character, the hardware requirements grew in two dimensions, the length and height. Each of those dimensions is linearly bounded as $O(m)$ but combined together yield a polynomial (quadratic) complexity $O(m^2)$. The actual worse case complexity is

$$Length * Height = (2 + 1.5m) * (2 + 0.5m) = 0.75m^2 + 4m + 4 = O(m^2)$$

Because the *Bus Match Line* value is held in the *Pattern Latch*, the maximum height the bus can obtain is |Σ|. No bound is placed on the length although the worst case analysis suggests that triple the height $\left(\frac{1.5}{0.5} = 3\right)$ of the bus might be a good minimum.

3.2.4.3. Converting a Right-Linear Grammar to Its Hardware Instantiation

This chapter has discussed the equivalence of the languages recognized by an FSA and a regular expression. Algorithms have been presented to convert either form into our hardware algorithm. Right-linear and left-linear grammars are additional methods of describing the same set of languages as those recognized by FSAs and regular expressions (Harrison 1978). A left-linear grammar is trivially converted to a right-linear grammar and visa versa. This section presents the conversion of a right-linear grammar to our hardware algorithm.

A grammar is defined in Barrett and Couch (1979) by a four tuple (Σ, N, P, S) where:

(1) Σ is a finite set representing the terminal alphabet.

(2) N is a finite set representing the nonterminal alphabet. Σ and N are disjoint.

(3) $S \in N$ and is the designated start symbol.

(4) P is a set of productions (rules) of the form y→x where y and x are in $(N \cup \Sigma)^*$ and y contains at least one element in N.

A right-linear grammar further restricts P to be of the form A→xB or A→x where A and B are in N, and x is in Σ. The 'cocoa' example from earlier in this chapter could be defined by the following grammar.

$\Sigma = \{a, c, o\}$

$N = \{A, B, C, D, E\}$

$S = \{A\}$

$P =$ {

A → cB
B → oC
C → cD
D → oE
E → a

}

One nonterminal may have several productions associated with it as is shown here in a right-linear grammar for the regular expression 'a(b|c)*d'.

$\Sigma = \{a, b, c, d\}$

$N = \{A, B\}$

$S = \{A\}$

$P =$ {

A → aB
B → bB
B → cB
B → d

}

Noting that a right-linear grammar contains a terminal on the right hand side of every production, conversion from the grammar to our hardware consists of:

(1) Assign bus line 0 to pattern found.

(2) Assign the start symbol to bus line 1.

(3) Assign all remaining nonterminals to unique bus lines.

For every production of the form $\alpha \rightarrow \beta\delta$, load three cells with α as a read-cell, β as a match-cell, and δ as a write-cell. If $\delta = \{\}$, then write to pattern found.

Applying these rules to the grammar for 'a(b|c)*d':

(1) Bus line 0 = pattern found.

(2) Bus line 1 = A.

(3) Bus line 2 = B.

(4) Productions:

	pattern	=	1 a 2
	read	=	1 0 0
A → aB ⇒	write	=	0 0 1
	fldc	=	0 0 0
	vldc	=	0 0 0

	pattern	=	1 a 2 2 b 2
	read	=	1 0 0 1 0 0
B → bB ⇒	write	=	0 0 1 0 0 1
	fldc	=	0 0 0 0 0 0
	vldc	=	0 0 0 0 0 0

	pattern	=	1 a 2 2 b 2 2 c 2
	read	=	1 0 0 1 0 0 1 0 0
B → cB ⇒	write	=	0 0 1 0 0 1 0 0 1
	fldc	=	0 0 0 0 0 0 0 0 0
	vldc	=	0 0 0 0 0 0 0 0 0

	pattern	=	1 a 2 2 b 2 2 c 2 2 d 0
	read	=	1 0 0 1 0 0 1 0 0 1 0 0
B → d ⇒	write	=	0 0 1 0 0 1 0 0 1 0 0 1
	fldc	=	0 0 0 0 0 0 0 0 0 0 0 0
	vldc	=	0 0 0 0 0 0 0 0 0 0 0 0

A very simple heuristic can recognize that adjacent routing cells are writing and reading the same bus line and thus collapse the redundant cells yielding:

pattern	=	1 a 2 b 2 c 2 d 0
read	=	1 0 1 0 1 0 1 0 0
write	=	0 0 1 0 1 0 1 0 1
fldc	=	0 0 0 0 0 0 0 0 0
vldc	=	0 0 0 0 0 0 0 0 0

In order to prove that no "nasty realities" stand in the way of actually implementing these hardware searching algorithms, an n-MOS implementation was designed and fabricated.

3.3. PAM - An Implementation of a Pattern Matching Chip

Because the paper design might not foresee all of the implications of realization, a project was initiated (Curry et al. 1983) to fabricate an n-MOS (grounded substrate - Metal Oxide Semiconductor) (Mead and Conway 1980; Mukherjee 1986) implementation of the hardware algorithm as defined for multiple-exact-patterns with wild cards. Indeed, the algorithm as implemented in this project is somewhat different from the algorithm definition in section 3.2.3.2. The experience gained from this implementation and, subsequently, the integration of the algorithm into the *String Coprocessor* as described in later chapters, lead to significant improvements for practical use of the searching algorithm. This section will not only describe the details of this implementation, but will also augment those details with improvements for better implementations.

At the time of this project, the algorithm was well understood for concatenation with wild cards, and the changes for multiple patterns (alternation with no parentheses) were conceived during the implementation. The extensions for parentheses and closure had not yet been derived. The technology of choice was n-MOS, but there is nothing related to the algorithm restricting it to any particular technology. Having selected n-MOS, though, leads to a preference to inverted logic and the first change noticeable in the basic cell replaces the *and-gates* and *or-gates* with *nand/nor* logic as shown here.

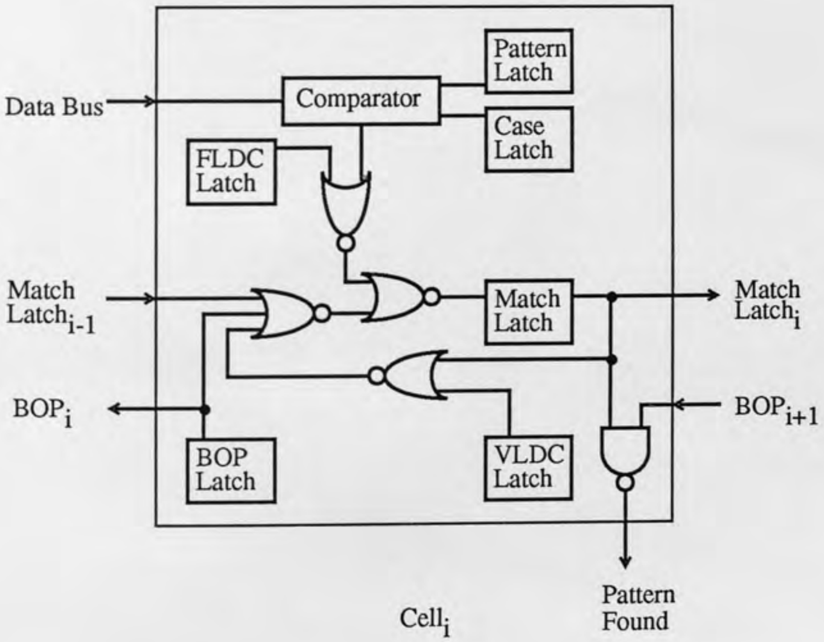


Figure 3.28. The Basic Match Cell with Inverted Logic

Two other differences are immediately evident as well. Rather than an *EOP Latch*, a *BOP Latch* (Beginning Of Pattern) is used to mark multiple pattern boundaries and a new latch is introduced labeled *Case Latch*.

The *BOP Latch* is functionally identical to the *EOP Latch* already known to the reader. It forces a 1 input on the *Match Latch*$_{i-1}$ input for the first character of a pattern and is used as an input to set the *Pattern Found* output. The *BOP Latch* was later replaced with the *EOP Latch* technique for two reasons. First, the *BOP Latch* is easy to determine when to set in a left-to-right preprocessing of the expression but the unary operators of regular expressions are on the right of their object, thus requiring the right-to-left preprocessing algorithm of section 3.2.3.2. In order to preprocess the *BOP Latch* technique from right-to-left, a look-ahead would be required to determine if the current character was on a pattern boundary or not. Second, if the pattern is left justified in the cells (i.e., if the total length of the pattern is less than the total number of hardware cells and if the pattern is loaded with the first pattern character in the first hardware cell), then an extra cell is required after the last character of the last pattern with the *BOP Latch* set in order to properly activate the *Pattern Found* logic. This is a result of having the *BOP Latch* information flow left between cells for the *Pattern Found* logic. The *EOP Latch* technique resolves all of these problems as do the *Read Latch* and *Write Latch* of the full regular expression algorithm.

The *Case Latch* was incorporated to facilitate searches which were case independent. For example, this dissertation labels figures with a capital "F" on the word "figure" but refers to these figures with a lower or upper case "f" depending on the sentence position. A regular expression to search for all occurrences of the word "figure" in this dissertation might be "(F|f)igure". Using the hardware of this implementation with 8 hardware cells, the pattern would be:

TABLE 3.14. PAM Instantiation of '(F|f)igure'

pattern	=	f	i	g	u	r	e	0	0
case	=	1	0	0	0	0	0	0	0
fldc	=	0	0	0	0	0	0	0	0
vldc	=	0	0	0	0	0	0	0	0
BOP	=	1	0	0	0	0	0	1	0

The *Case Latch* works by disabling the comparator for the bit which indicates the case of the character. That is bit 6 in both ASCII and EBCDIC where bit 0 is the least significant bit. The hardware can be used for any alphabet by simply fixing the *Case Latch* to the last bit of the comparator and then externally connecting the input bits appropriately so that the character set of the machine has the case bit placed last. This algorithm allows case control on a character by character basis, but this entire latch was later dropped from the algorithm since most searching algorithms simply convert the entire data stream to lower case if case independence is requested for the search. Individual character case independence can be specified in the regular expression algorithm through a parenthesized alternation as shown in the "(F|f)igure" example.

The diagrams of the basic cell have left the latches and the comparator as undefined boxes of logic. Because of the complexity of the timing associated with the data flow for loading the pattern and the inter-cell communication, the clocking of the data and components must be thoroughly defined. A simple means to accomplish the delay timing required in the latches would be to have two inverters in series with non-overlapping clocks gating their input. The timing of the *Pattern Latch* needs to be out of phase with the *Match Latch* and *Data Bus* in order to sequence the state information with the next input character. The following diagram expands the definition of the latches for this algorithm.

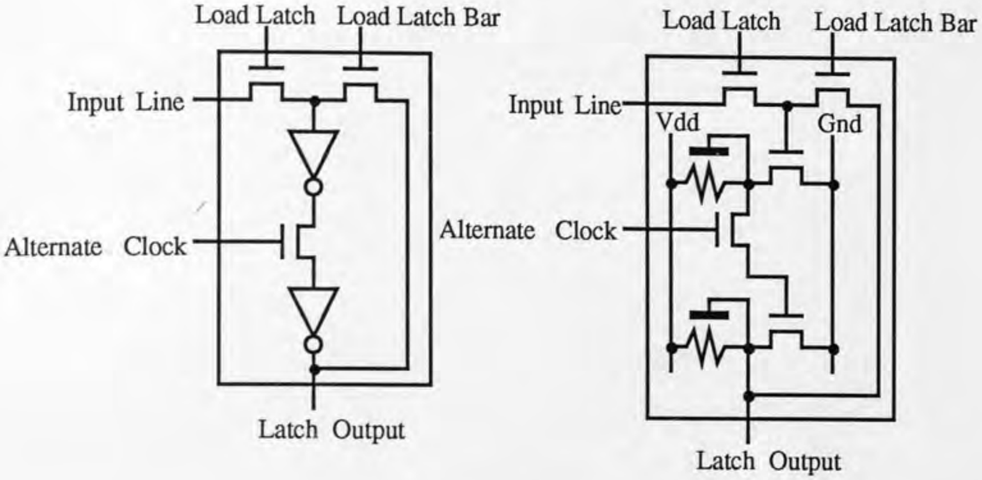


Figure 3.29. n-MOS Latch Implementation

The design as shown in the diagram provides all of the logic necessary for an n-MOS implementation. The diagram on the left utilizes a notation mixing pass transistors and gate logic. The diagram on the right expands the gate logic to its transistor level implementation. Other technologies may prefer to utilize different techniques to accomplish the same logic. As long as the timing and combinational logic between latches remains consistent, the algorithm will remain correct. For consistency throughout this section, all logic will be considered to be implemented in n-MOS and all logic diagrams will utilize the schematic approach on the left.

Because signals will dissipate in the latch, it must continually refresh itself. The output of the second inverter is fed back into the first inverter. Alternatively, a new value may be loaded into the latch from the *Input Line* to the first inverter. The *Load Latch* and *Load Latch Bar* control lines are complementary of each other but are both qualified with a clock phase. The clocking is based on a two phased, non-overlapping clock scheme. Phase one will be labeled $\phi 1$ and phase two $\phi 2$. If the "load control lines" are qualified with $\phi 1$, then the *Alternate Clock* is $\phi 2$. If they are qualified with $\phi 2$, then the *Alternate Clock* is $\phi 1$. Since $\phi 1$ and $\phi 2$ are non-overlapping, all logic has a chance to stabilize in the feedback path. The following timing diagram shows the relationship to the clocks for all control lines during a refresh cycle and a load cycle.

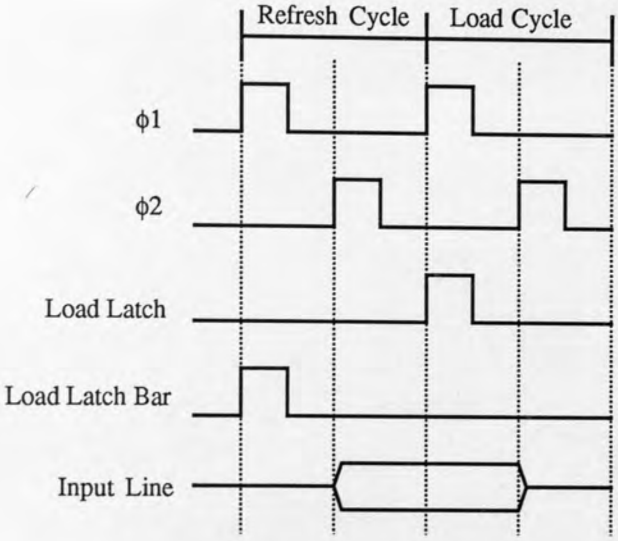


Figure 3.30. Latch Refresh/Load Timing

This particular timing diagram shows the "latch load phase" synchronous with ϕ1 and "input change phase"/"alternate clock" synchronous with ϕ2. Some latches will be timed this way and some will be timed with ϕ2 and ϕ1 respectively. However, the logic still remains the same. One other point to note in this design is the fact that the latch can be loading a new value on its input while retaining its old value on the output during the first phase of the clock. This is the key to the ability to simultaneously shift the pattern through all cells during the pattern load sequences, and it also provides the means to perform the *Match Latch* delay by keeping the *Match Latch* and *Data Line* in phase with each other and the *Pattern Latch* in the alternate phase.

A convenient modification to the latch, which was determined from the results of this implementation, adds one more control line and replaces the first inverter with a *nand-gate*. The control line will be used to clear the latch of its contents. The control line is active low and synchronized with the input phase of the clocking. A schematic and timing diagram are shown in the following illustration.

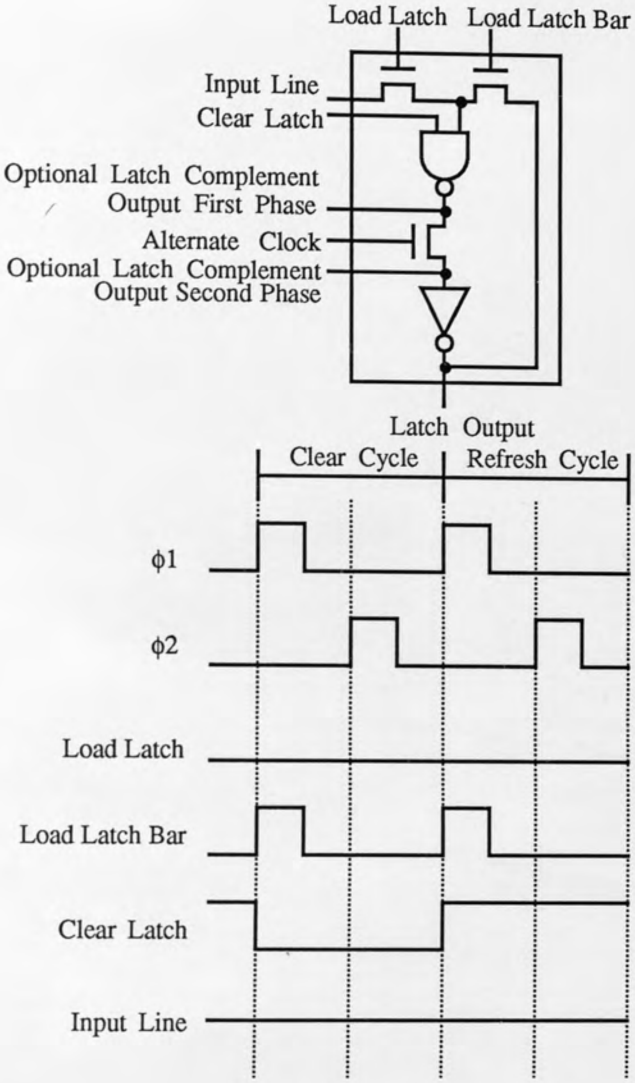


Figure 3.31. Clearable Latch Schematic and Clear/Refresh Timing

When *Clear Latch* is a logic 0, the *nand-gate* output is always a logic 1. If *Clear Latch* is a logic 1, then the *nand-gate* output is the inverse of the *Latch Input*.

The implementation of the algorithm did not include the *Clear Latch* control line and therefore always had to shift in null pattern entries to flush the old pattern. This tied the preprocessing complexity to the number of hardware cells rather than the pattern length. Being able to simultaneously clear all cells with a control line is a simple resolution which places the preprocessing complexity back with the pattern length. With the single exception of the *Match Latch*, all latches used in the rest of this section will be described using exactly the design and timing of Figure 3.31. The *Optional Latch Complement* outputs are potentially useful values that can be utilized if required. The timing of any particular control line in a diagram will be specified by appending "$*\phi 1$" or "$*\phi 2$" denoting qualification with the respective clock phase, or by providing an explicit timing diagram.

Limiting the match cell to exact-pattern-matching and the alphabet size to two elements (i.e., $|\Sigma| = 2$), the following logic would be a complete instantiation of the hardware and control lines necessary for a two-bit pattern.

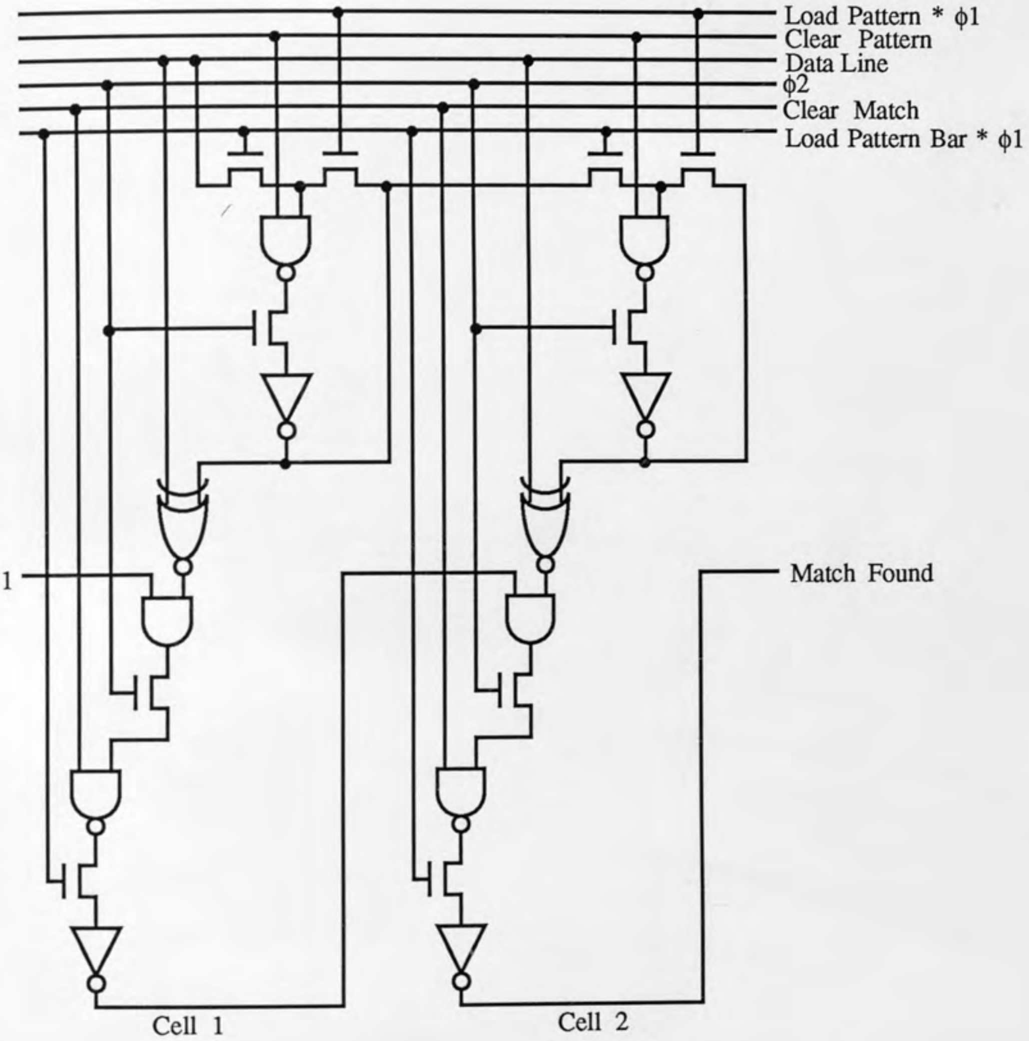


Figure 3.32. Fully Instantiated Two-Bit Pattern Hardware

At this point in time, the ϕ2 clock qualification into the *Match Latch nand-gate* is not actually necessary but helps to clarify the timing and will be necessary later when the *VLDC Latch* logic is included.

Only a small portion of the hardware is involved in loading and retaining the pattern. Just the *Pattern Latch* hardware is shown in the following diagram.

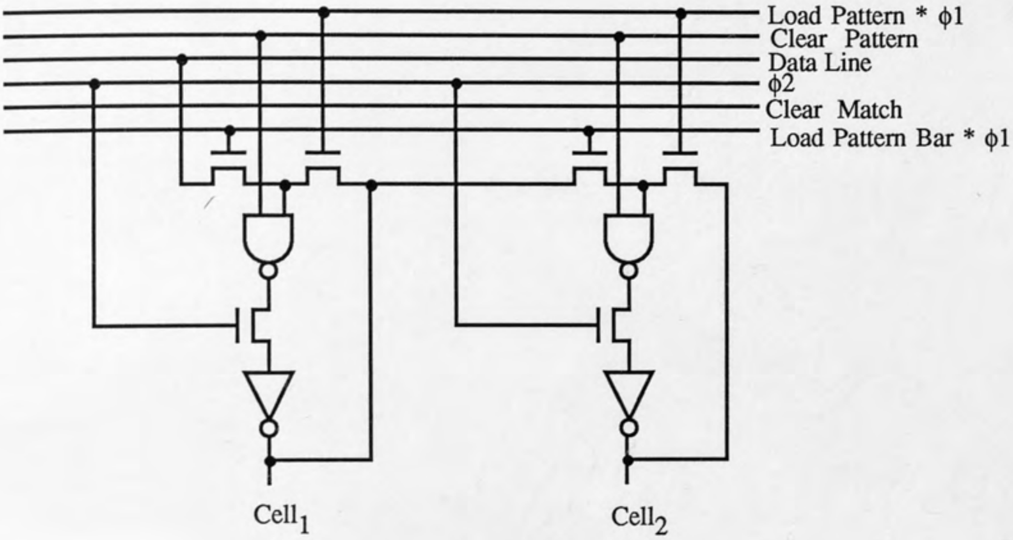


Figure 3.33. Pattern Latch Hardware Only

The sequence to load a new pattern would be to clear the pattern in all cells first and then sequentially place the pattern on the *Data Line*, shifting it in one bit at a time. The timing to clear the cells and shift in the first two pattern bits is shown in the following diagram.

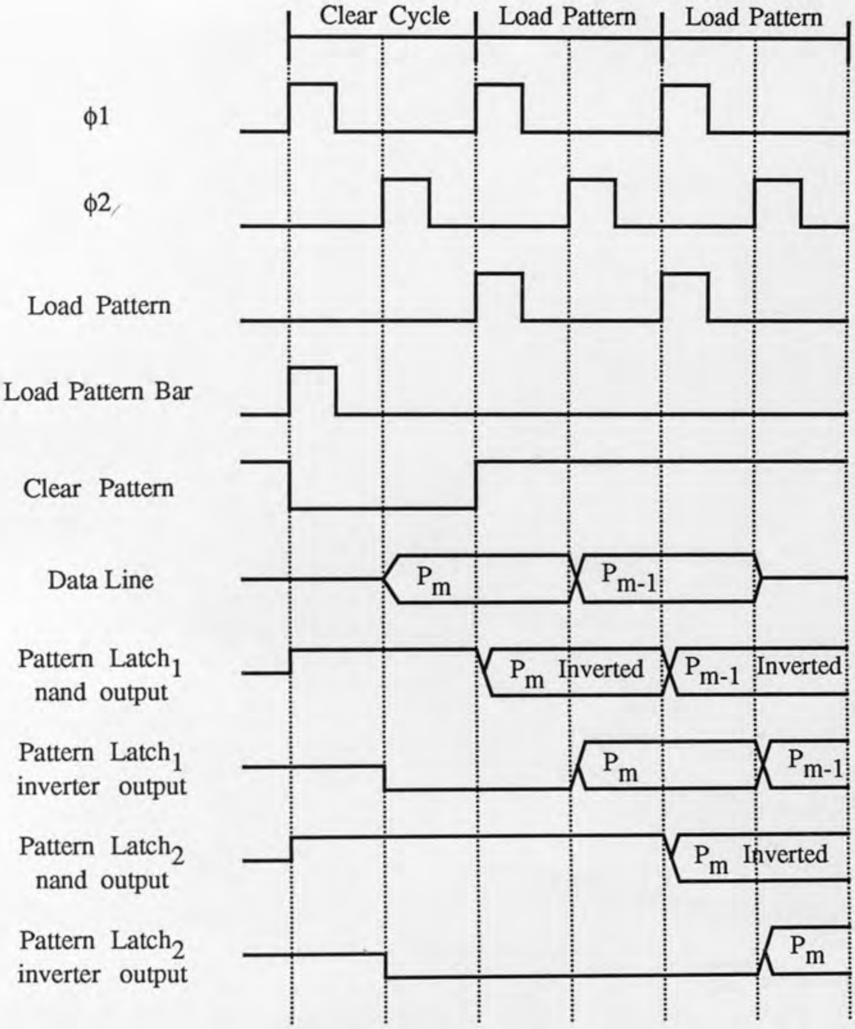


Figure 3.34. Clear and Load Pattern Timing

As shown, the *Clear Pattern* control line dropped to a logic 0 will force the latch to be logically holding a 0. The *Clear Pattern* control line is broadcast to all cells and held low through an entire $\phi 1$, $\phi 2$ cycle. This allows the "clear" to propagate through both the *nand-gate* and *inverter* of all of the *Pattern Latches* simultaneously. Coincident with the $\phi 2$ cycle of the "clear", the data bit of the pattern is placed on the *Data Line*.

The first $\phi 1$ after the "clear" starts the "load" and the *Data Line* propagates to the *nand-gate* of *Pattern Latch*$_1$ while the *inverter* output of *Pattern Latch*$_1$ is propagated to the *nand-gate* of

Pattern $Latch_2$ and so on. During ϕ2 of the "load", the next pattern bit is loaded on the *Data Line* preparing for another shift to occur on the next ϕ1. This is continued until the entire pattern is loaded, leaving the hardware ready to begin searching.

Once the pattern has been loaded, the search can actually begin. This is accomplished by first insuring that all of the match latches are set to a known state of logic 0 and then broadcasting each bit of the data to be searched to all of the cells. The *Load Pattern* control line will always be a logic 0 throughout the search and the *Clear Pattern* control line will always be a logic 1. Hence, the *Pattern Latches* will simply be refreshing themselves during the entire search. The timing for control lines and combinational logic is shown below.

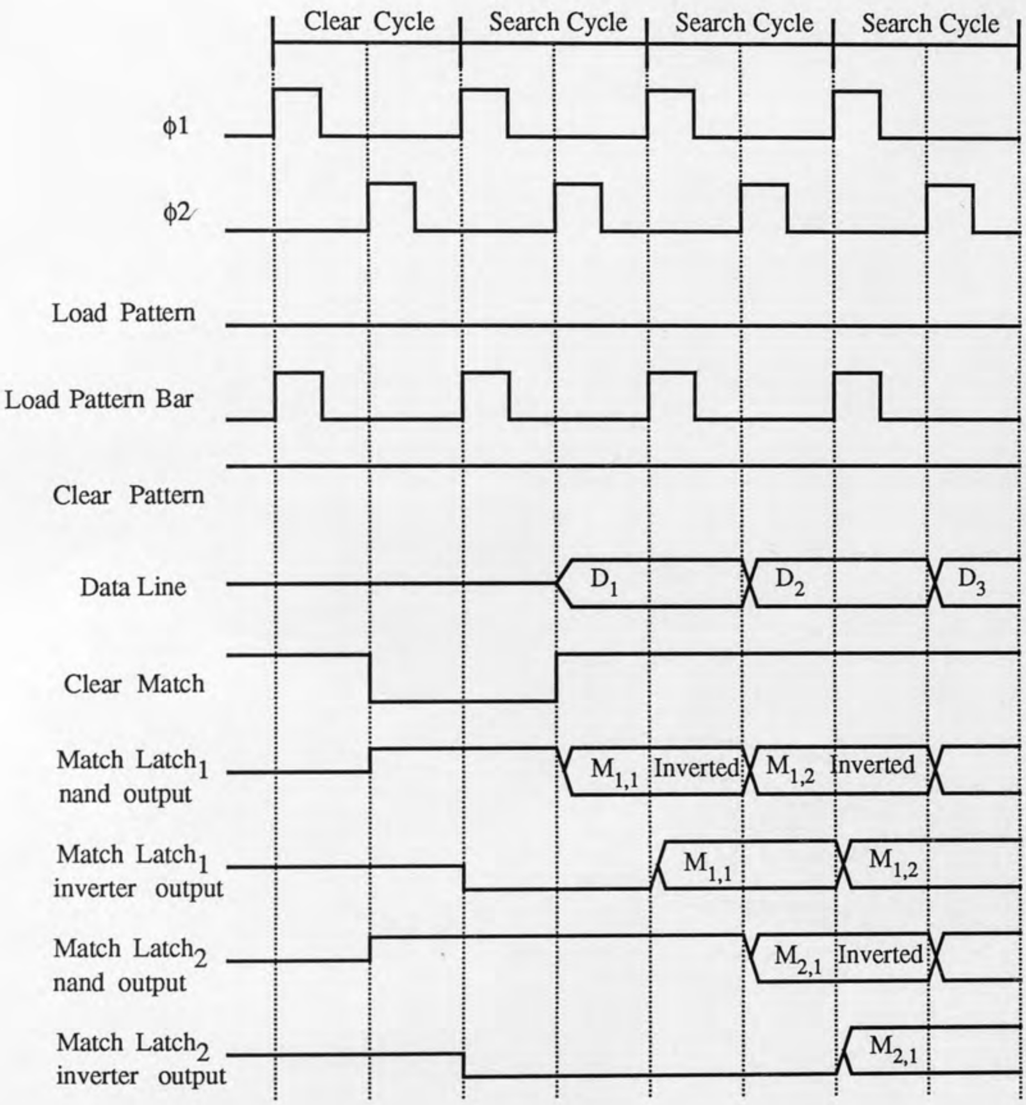


Figure 3.35. Timing of a Search

The *Clear Match* line is broadcast to all cells and held low during an entire φ2, φ1 cycle. This forces zeros into all of the *Match Latches* simultaneously, thus insuring that no cell can contain any information that indicates a successful match when the search has not yet begun. Once the search has begun, a new data item must appear on every φ2 cycle until the data is exhausted.

Note that the *Match Latch* inverter output **does** coincide with the start of the next search cycle and remains stable.

The expansion of the *Data Line* into a *Data Bus* implies that the *Pattern Latch* must be expanded to hold an entire data item in it. This also implies that the *exclusive-nor* used as the comparator needs to be expanded to hardware which properly accommodates *2b* inputs (where *b* is the number of bits in the *Data Bus*) and still outputs the appropriate single bit result indicating if D_k (the k^{th} data item) and P_i (the i^{th} pattern item) are identical or not. These changes are actually quite simple to implement.

By combining *b* separate latches together in one cell and connecting them as shown in the following diagram, the timing, logic, and control lines remain identical to the original cell implementation for the *Pattern Latch.*

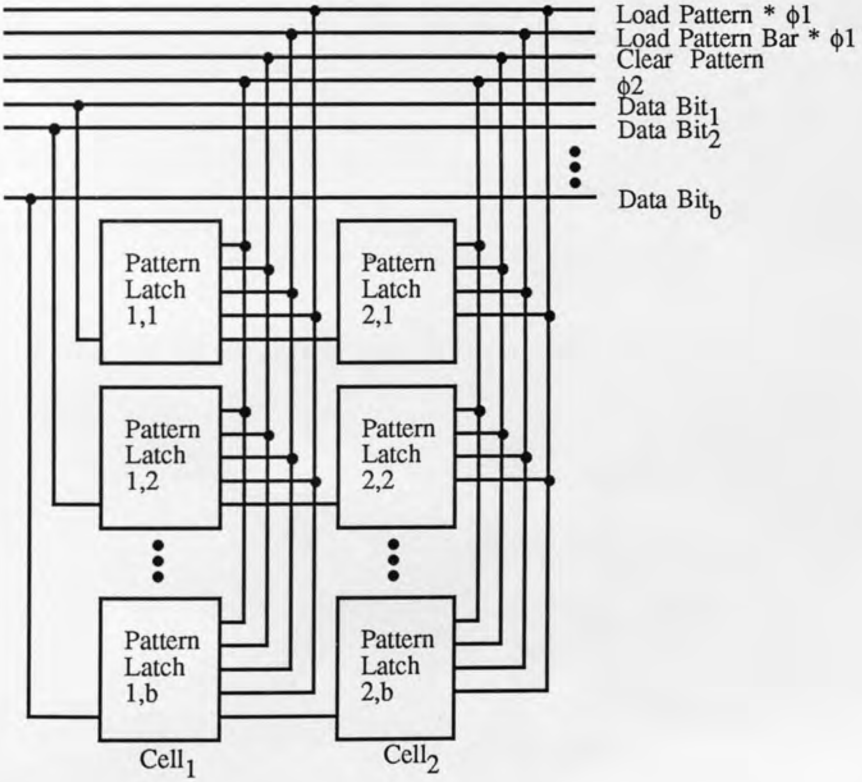


Figure 3.36. Pattern Latch for a Data Bus

As mentioned, the control lines and their timing are unchanged. The difference is that they are broadcast to $b*m$ latches instead of m latches. Each of the b latches in the column is identical in timing and logic to the *Pattern Latch* defined earlier. A column of b latches now constitutes a single *Pattern Latch* with no communication in the column. A row of m latches is identical in all aspects to the *Pattern Latch* hardware implementation previously defined where communication is limited to the immediately adjacent cell to the right.

The notation *Pattern* $Latch_i$ will still be used to denote the entire *Pattern Latch* in $Cell_i$ and the notation *Pattern* $Latch_{i,j}$ will now refer to an individual bit in a *Pattern Latch.* The variable i will denote which cell the *Pattern Latch* is in and is bounded by $1 \leq i \leq m$. The variable j denotes which bit in the *Pattern Latch* of $Cell_i$ is being referred to and is bounded by $1 \leq j \leq b$.

A pattern load sequence:

(1) Clears all *Pattern Latches* using the *Clear Pattern* control line as previously defined.

(2) Places the pattern data item on the *Data Bus* with bit-one on *Data* Bit_1, bit-two on *Data* Bit_2 and so on through bit-b on *Data* Bit_b all simultaneously set.

(3) Shifts the *Data Bus* contents into *Pattern* $Latch_1$ while simultaneously *Pattern* $Latch_1$ shifts into *Pattern* $Latch_2$ and so on.

In effect, the pattern loading sequence has remained identical with the single exception that there are b rows of pattern latches rather than one row. Once the entire pattern is loaded, the contents of the *Pattern Latches* becomes static and searching can begin. The changes to the *exclusive-nor* hardware must now be considered.

Because of the nature of n-MOS designs, the expanded *exclusive-nor* implementation in that technology is fairly uncomplicated. The *exclusive-nor* is defined by the following logic:

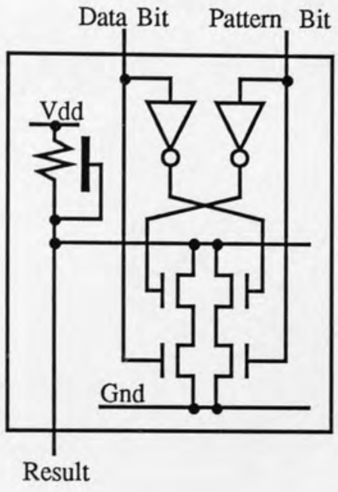


Figure 3.37. Two-Input Exclusive-Nor n-MOS Implementation

The result line is pulled high to a logic 1 through the depletion mode transistor (resistor to Vdd). This line can potentially be pulled down to a logic 0 through two paths to ground (Gnd). If the *Data Bit* and *Pattern Bit* values are the same, then no path to ground exists and the result line will remain a logic 1. If the two bits are different, then one of the paths to ground will be opened and the result line is pulled down to a logic 0.

Since each bit in the *Pattern Latch* has an "optional inverted output" in its logic, the inverter for the *Pattern Latch* input to the *exclusive-nor* is redundant and unnecessary. Furthermore, since the *Data Bit* is being broadcast to all cells, it would be more efficient to broadcast its inverted value as well, thus requiring only one inverter rather than *m* inverters. To expand this logic for *2b* inputs, more potential paths to ground are provided as shown by the following diagram.

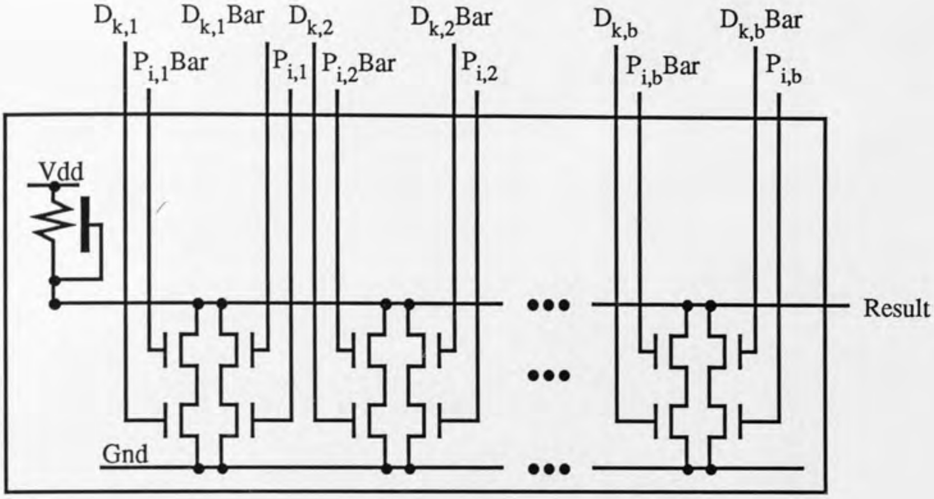


Figure 3.38. 2b-Input Comparator n-MOS Implementation

This diagram presumes that the *Data Bits*, the *Pattern Bits*, and their respective complements are generated externally to the *comparator*. The label $D_{k,1}$ refers to the first bit of the k^{th} data item in the string being searched. $D_{k,1}$ *Bar* is the complement of that bit. $P_{i,1}$ is the respective bit in *Pattern Latch*$_i$ and $P_{i,1}$ *Bar* is the *optional latch complement output, second-phase* from that same latch. For every bit j where $1 \leq j \leq b$, $D_{k,j}$ and $P_{i,j}$ are compared and any differences provide a path to ground producing a logic 0 on the *result* output. All bits are compared simultaneously and multiple paths to ground may occur. If there are no differences, then no path to ground is available and the result is left a logic 1.

This result is not the same as a *2b-input exclusive-nor*, but rather there are *b exclusive-ors* which are *nor-ed* together yielding a correct result for a comparator.

Given the repetition of the *Pattern Latch* bits and the *Comparator* logic, a natural hierarchical cell layout definition would start by combining the *Pattern Latch* and *Comparator* into one cell. Such a layout is shown in the following diagram.

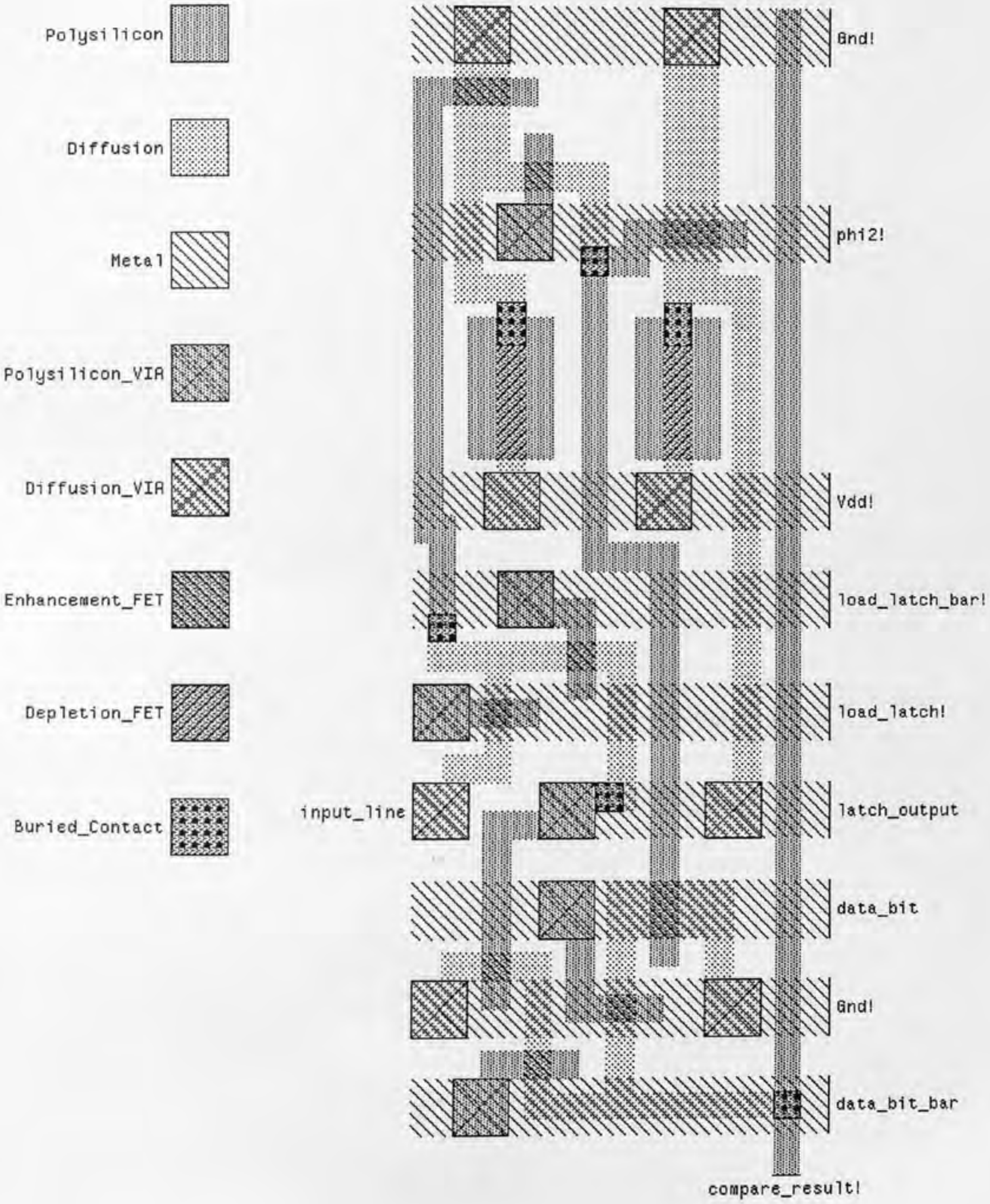


Figure 3.39. Pattern Latch$_{i,j}$ and Comparator D_k

The stipple patterns for the various n-MOS layers are labeled and defined down the left side of the figure. Any label that ends with an exclamation mark (!) is a global label that is presumed to be later connected to all other similarly labeled signals. All labels are identical to those of Figure 3.29. The trivial clear-pattern improvement requires routing another control line horizontally and changing the first inverter to a nand gate. Vdd and Gnd are the power lines. Phi2 is the ϕ2 clocking signal which is the alternate clock for this latch. The *load_latch, load_latch_bar* and *input_line* signals, are all clocked identically to the timing diagram of Figure 3.30. Additionally, the data bit and its complement are passed through this cell and *exclusive-nored* directly with the *Pattern Latch* bits.

In order to implement the logic associated with the *Case Latch*, one of the *exclusive-nors* must have three inputs to it. That special case layout is shown in the following diagram.

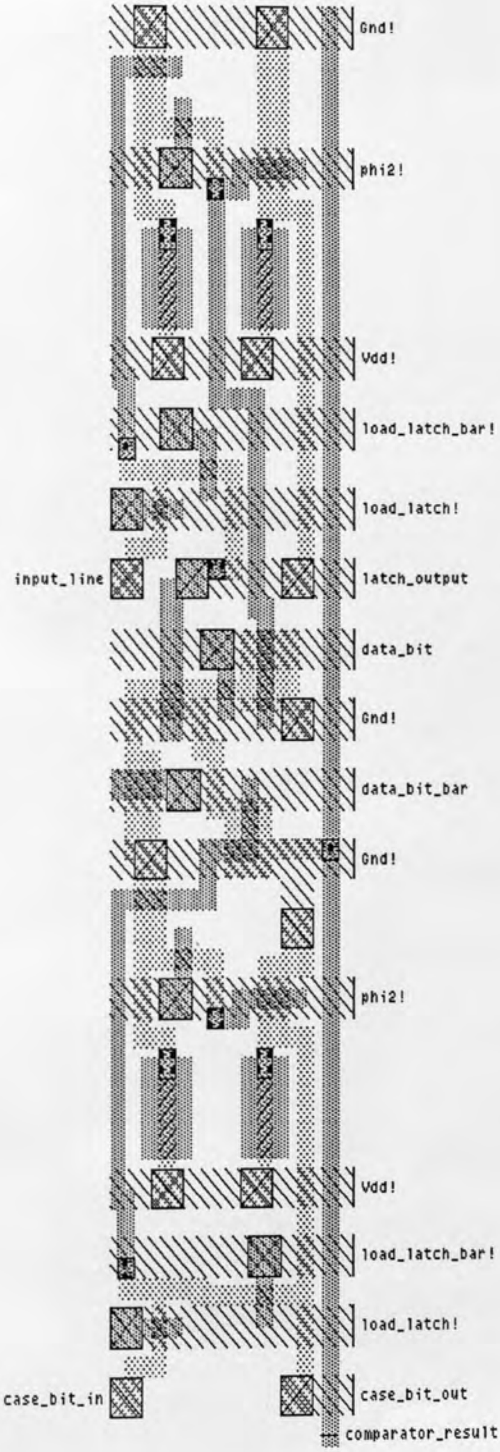


Figure 3.40. Pattern Latch and Comparator with Case Latch

In this implementation, the alphabet size was set to be 256 characters, thus covering both of the industry standard character sets ASCII and EBCDIC. Binary encoding of 256 characters implies 8 bits ($log_2\ 256 = 8$). Seven of the cells as shown in Figure 3.39 are stacked vertically on top of one of the cells as shown in Figure 3.40 to form a byte. A pull-up resistor (depletion mode transistor) is connected to the *compare_result* line at the top of this vertical stack and the remainder of the latches and their logic are connected below this stack as shown in the next diagram. The cell is 30λ wide (where λ is a constant related to the width of lines measured in microns) by 1077λ tall for a ratio of 1:36. The *Pattern Found* logic adds another 515λ to the height making the aspect ratio 1:53. This aspect ratio is intentional since 64 of these cells will be placed in a horizontal array, and Input/Output pads and signal amplification logic contribute to the overall width. All detail is lost when plotted at this scale, but the aspect ratio is dramatically illustrated.

Figure 3.41. One Complete Match Cell

The following schematic shows a rough floorplan of the single match cell. The 8-bit *Pattern Latch, Comparator,* and *Case Latch* account for two-thirds of the height of the cell. The remaining one-third of the height is a layout of the following logic.

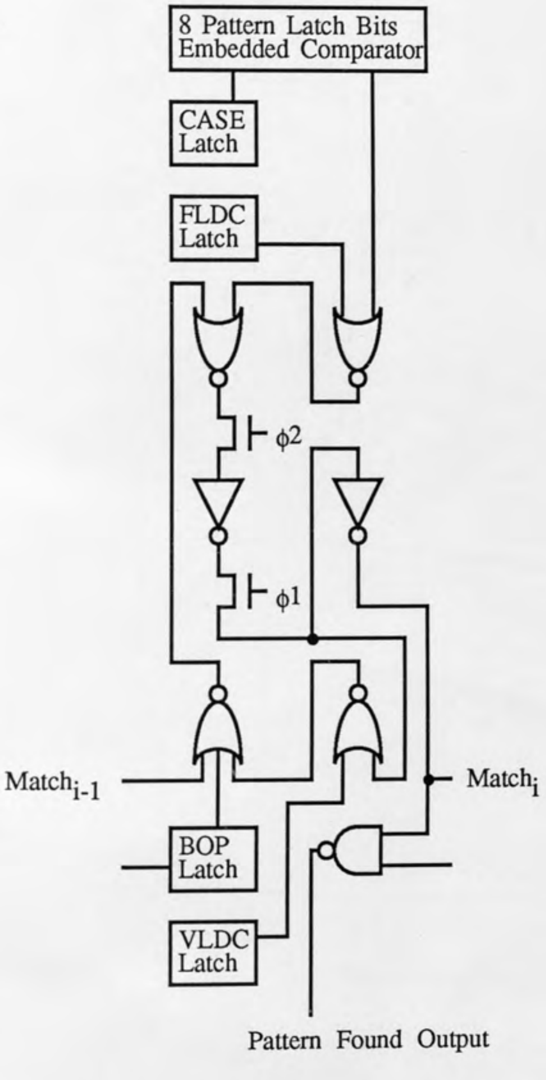


Figure 3.42. Schematic of the Match Cell Implementation

The *Pattern Found* logic consists of a slightly modified encoder which informs the outside world which of the 64 cells currently has a *Pattern Found* set active. For a binary encoding, 64 cells implies 6 bits (*log_2 64 = 6*) but two problems exist. One, 6-bits **does** give 64 combinations of output, but what if no cells currently have *Pattern Found* set active? Second, what happens to the encoder logic if more than one cell has *Pattern Found* set high? The first problem is resolved by one more output bit which is used to indicate if **any** *Pattern Found* lines are active. Thus, if that bit is set to 0, then the address bits should be ignored by the outside world. If that bit is set to 1, then the address bits indicate the right-most cell which has *Pattern Found* set active. The implementation also adds another information bit which informs the outside world if multiple matches occurred. The encoder solves the second problem by having any cell with an active *Pattern Found* disable all cells less than it from setting the address bits. A 2-bit encoder for four cells is shown here to illustrate how this priority scheme is implemented.

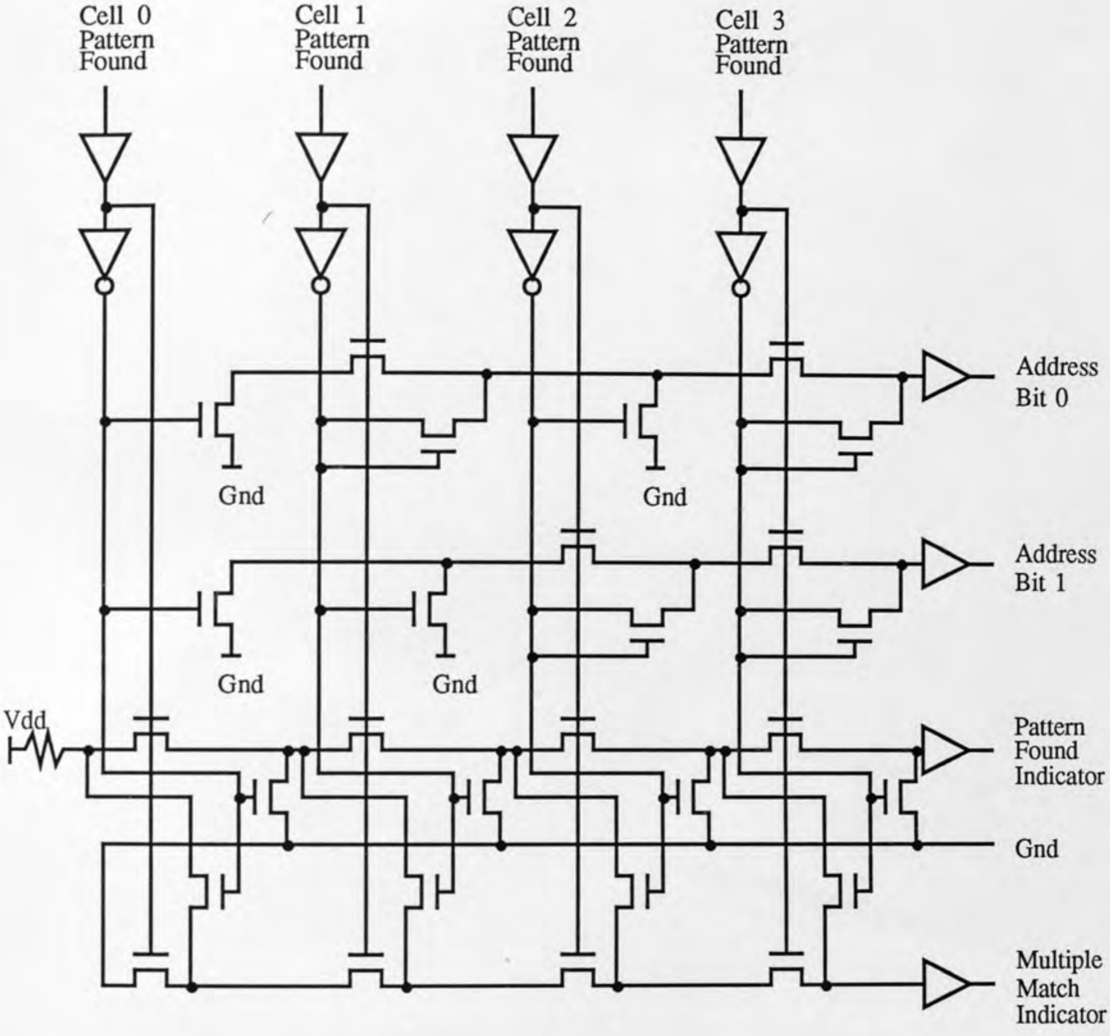


Figure 3.43. 2-Bit Right-Most Priority Encoder

The encoder does a straight binary mapping of the cell ordering where the first cell address is 00 and the last cell address is 11. Because the *Pattern Found* output of each cell is a *nand-gate*, the line is active low. If a cell needs to ground an address line making it a 0, then the line is pulled low and it does not matter if any other cells also ground it. If a cell needs to set an address line high, then it needs to make sure no cell to its left grounds the line. Thus, the pass transistors are employed as seen in each location where an address line should be set high.

The encoder of the implementation expands this logic to 6 bits for the 64 cells. The *Pattern Found Indicator* and *Multiple Match Indicator* remain unchanged in the implementation at the bottom of the encoder. Each set of four cells has a non-inverting amplifier (super buffer) precisely as shown in Figure 3.43.

The 64 cells plus the encoder are surrounded by Input/Output pads and control logic. The following plot shows an entire chip rotated ninety degrees. The stipple patterns have been replaced by a dithering algorithm to help enhance the visibility for this low resolution, monochrome medium.

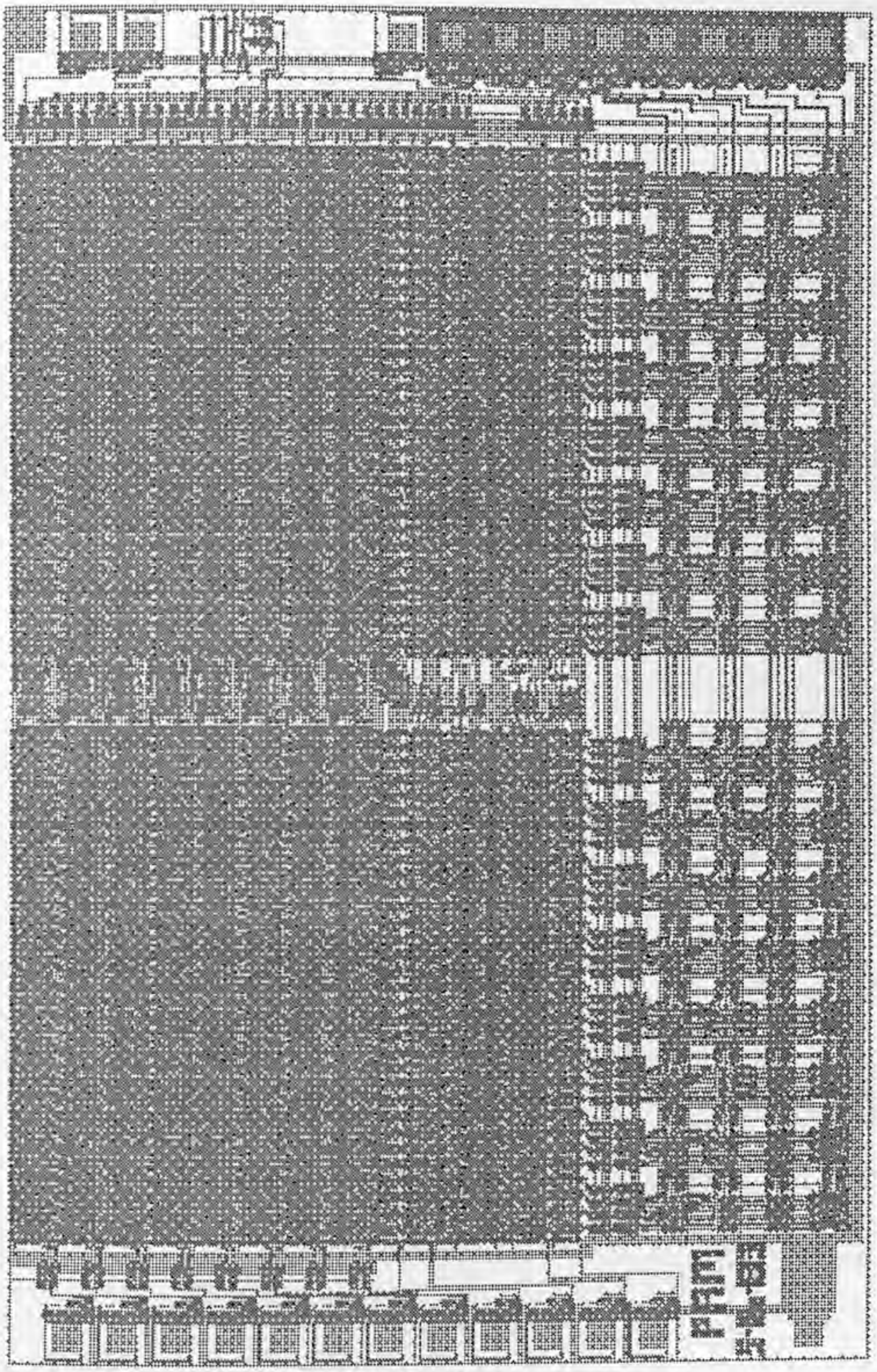


Figure 3.44. Plot of Entire Pattern Matching Chip

The thirteen pads across the bottom of the plot from left to right constitute the eight pattern bit inputs; the *fldc, case, bop,* and *vldc* inputs; as well as the Gnd pad. The pads across the top of the plot are Vdd, $\phi1$, *load/search*, $\phi2$, the six *Pattern Found* addresses, the *Pattern Found Indicator*, and the *Multiple Match Indicator*. The eight pattern input pads are used to load the pattern during preprocessing and to broadcast the data bits during the search. The load/search pad is used to denote whether the chip is in the pattern load phase (preprocessing) or in search mode. It is qualified with $\phi1$ and $\phi2$ and then broadcast appropriately as *load_latch* and *load_latch_bar*. All signals which are broadcast to the entire chip are amplified on the front end of 32 match cells and then restored again for the next 32 match cells.

The design was fabricated and thoroughly tested for function, speed, and power consumption. We received several chips from two different foundries and found dramatic variance. The best results yielded complete functionality, the ability to search at 2.5 million characters per second, and drew approximately 140 milliamperes. These results were pleasing since they were comparable to semiconductor memory and CPU speeds of the time and were well beyond common disk transfer rates. The concept of a silicon subroutine for searching at memory speeds and the suitability of the algorithm for realization was thus proven viable.

3.4. Design Alternatives

As already mentioned, the experience of the implementation led to some differences in the final definition of the algorithm. The use of the *Clear Pattern* and *Clear Match* control lines are definitely recommended for any implementation of the algorithm. There are some other changes that might be considered depending on the use of the algorithm.

One limiting factor is the number of match cells that can physically be placed on a single chip. While chip densities are continually improving, one can always imagine a scenario where more match cells might be required than can fit on a single chip. The algorithm does not provide for the situation where the preprocessed pattern length exceeds the number of physical cells. A possible solution would be to interconnect several of the pattern matching chips in series. For all of the algorithm versions except full regular expressions, multiple chip connections would only require one additional input pad for the *Match Latch*$_1$ input. A 1 would be forced on the *Match Latch*$_1$ of the first chip and the *Pattern Found* logic, which already has outputs from the chip, can be used to appropriately set the *Match Latch*$_1$ input of the next chip. In the situation where only one of these chips is used, no speed penalty is imposed since the new input pad has a constant value connected to it. When multiple chips are used, a minor time penalty is paid for the inter-chip communication but that penalty is constant regardless of the number of chips since communication is limited to adjacent chips.

The regular expression algorithm would be difficult to expand to a multiple chip version since the entire communication bus would have to broadcast through all chips bidirectionally. The design of such a pad would be difficult, the number of pads would be limiting, and the performance would degrade further with each additional chip.

A change to the preprocessing algorithm can implement another common regular expression operator. A superscript plus mark ($^+$) is often used to denote "one or more" of a subpattern as opposed to the "zero or more" represented by a closure. This is called *iteration* and is quite commonly used in language theory but seldom found in searching algorithms. While the same expression could be defined by having one instance of the subpattern followed by another instance in closure, iteration is a convenient shorthand. Additionally, unlike closure, this operator can be integrated into the multiple-exact-patterns-with-wild-cards algorithm. This is accomplished by setting the *VLDC Latch* to 1 in the same cell as the iterated character. For example, the pattern P = "a^+b" represents the infinite set of strings that start with one or more "a"s followed by a single "b". This would be placed in the hardware matching cells as:

TABLE 3.15. Instantiation of 'a$^+$b'

input pattern	=	a^+b	
pattern	=	a	b
eop	=	0	1
fldc	=	0	0
vldc	=	1	0

The *VLDC Latch* serves as a feedback for *Match Latch*$_{i-1}$ but the comparator must still match the current character on the data bus before a match can occur in the cell.

Iteration can also be implemented in the regular expression algorithm rather easily. Single character iteration can be accomplished identically to the method described above since the *VLDC Latch* is still included in the regular expression algorithm. The iteration operator must immediately follow a close-parenthesis to have a larger scope than one character. The following rule can be applied to implement iteration. As before, the latches are shown as *Pattern Latch, Read Latch, Write Latch, FLDC Latch,* and *VLDC Latch* from top to bottom and the variables *Next, Prefix,* and *Suffix* have the same definition.

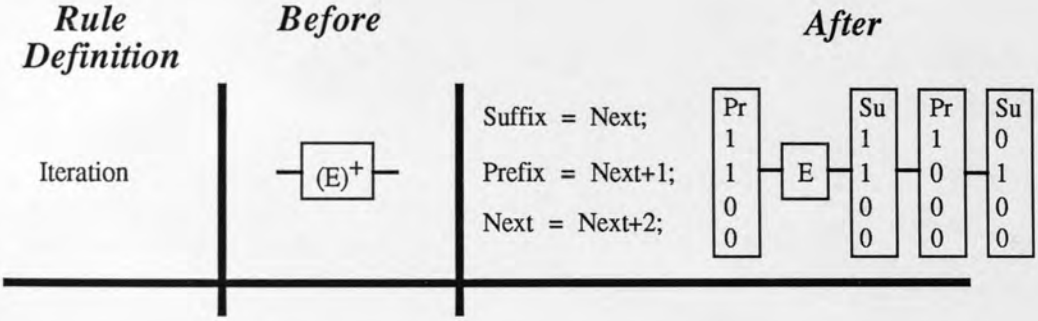


Figure 3.45. Rule for Iteration

Following are two examples of expressions utilizing iteration and their instantiations in the regular expression algorithm.

TABLE 3.16. Instantiations of 'ab^+c' and '$a(b|c)^+d$'

input pattern	=	ab^+c				
pattern	=	1	a	b	c	0
read	=	1	0	0	0	0
write	=	0	0	0	0	1
fldc	=	0	0	0	0	0
vldc	=	0	0	1	0	0

input pattern	=	$a(b\|c)^+d$											
pattern	=	1	a	3	b	2	3	c	2	3	2	d	0
read	=	1	0	1	0	0	1	0	1	0	1	0	0
write	=	0	0	1	0	1	0	0	1	1	0	0	1
fldc	=	0	0	0	0	0	0	0	0	0	0	0	0
vldc	=	0	0	0	0	0	0	0	0	0	0	0	0

Because iteration is a shorthand for a more complex operation, the complexity of the algorithm must be re-examined. Recall that single characters in closure accounted for the worst case complexity and that iteration is a shorthand for a subpattern repeated in closure. Then the example pattern "$a^+b^+c^+$" would be the same as _"aa*bb*cc*"_. It would appear that this would add more cells to our worst case behavior, but because the *VLDC Latch* can be employed, the length is actually lessened and the height is not used at all by iteration of a single character. Subpatterns in parentheses that are iterated do require two more cells than a normal parenthesized subpattern, but no additional height in the bus is required. The algorithm remains $O(m^2)$.

Another alternative we suggest that might be potentially useful is modification of the comparator. Currently it produces a binary answer of equal or not-equal. For some applications, such as back-end text retrieval architectures (Mukhopadhyay 1981), more comparison values could be quite useful such as *Pattern Latch* greater-than or less-than the current data value.

Different applications have different requirements and priorities for their searching algorithm. The next section discusses some of these requirements and compares our algorithm against several other published searching algorithms.

3.5. Comparison of Searching Algorithms

The introductory text of this chapter defined the attributes of a searching algorithm, briefly mentioned the various capabilities of several algorithms, and then described certain factors to review when selecting a searching algorithm. This section will delve into these subjects in much greater detail.

All of the searching algorithms have the goal of finding the pattern *P* in the data *D*. The general assumption is that *D* is very large. Sizes in the gigabytes and possibly terabytes could be reasonable for some databases. Not all searching algorithms can find **all** of the occurrences of *P* in *D*. Not all searching algorithms assume the data to be unstructured. We will only examine searching algorithms that do not require the data to be sorted or indexed. We will, however, mention two algorithms which assume the data to be in blocked record sizes.

This section will discuss the criteria on which the algorithms will be compared, present the algorithms discerning their attributes against the defined criteria, and then summarize these comparisons.

3.5.1. Searching Algorithm Comparison Criteria

The first criterion concerns the order in which the data is accessed during the search. If all of the data is available in random access memory (RAM), then the access order is not a real factor. If the data is in a memory cache, a paged virtual memory system, or streaming off of a peripheral storage device, then the access order **does** play a role.

Paging schemes for cache and virtual memory can incur a fair amount of overhead if frequent accesses occur across a page boundary. If the algorithm is to be used in a back-end architecture directly attached to a peripheral storage device such as a disk or tape unit, then backing up in the data stream becomes very costly in time and efficiency.

Another criterion is the complexity of the pattern specification and its effect on the complexity of the preprocessing and run-time complexities which are two more criteria for comparison. While some applications may only require *exact pattern matching*, others may require wild-cards or expression operators.

When comparing the preprocessing and run-time complexities, the order of complexity is certainly the first distinguishing feature. However, if two algorithms have the same order of complexity, then the magnitude of the coefficient should be examined. For example, if algorithm *A* has a run-time complexity of $O(n^2)$, algorithm *B* $O(n)$, and algorithm *C* $O(n)$, then it appears that algorithms *B* and *C* are both equivalent choices over algorithm *A*. However, if algorithm *B* is really *100n* and algorithm *C* *10n*, then algorithm *C* is clearly an order of magnitude faster even though both algorithms are $O(n)$. All algorithms will consistently be compared with the variables *m* and *n* representing the lengths of the pattern specification and the data respectively.

One other criterion that we introduced is the flexibility for redefining the data element size. We bring this up because of the potential ease that a software algorithm may have for making such changes and the relative difficulty for a hardware algorithm.

We now present the algorithms and their relationship to the criteria just defined.

3.5.2. Software Searching Algorithms

Software algorithms have the distinct advantage of flexibility. Most modern computing environments provide a memory hierarchy that can consist of registers in the CPU, high speed cache, lower speed RAM, virtual memory swap space on disk, and file storage on secondary medium. All of these resources are conceivably accessible to a software algorithm to dynamically employ as required. Most of the hardware algorithms are considerably less dynamic and must make use of the designed internal resources.

Further flexibility is represented by the ability to simply change and recompile a program for new features or changes in definition. Hardware is considerably harder to modify, especially at the chip level!

The trade-off for this flexibility is speed and resource requirements. As emphasized during the presentation of our algorithm, the complexities of the software algorithms grow to exponential size wheras there are hardware algorithms that are bounded by a simple quadratic. Furthermore, the software algorithms require more instructions during execution, each of which is kept in memory along with the data. This leads to the hardware algorithms being tremendously faster.

The software algorithms are now presented with analysis of their various complexities.

3.5.2.1. The "Intuitive" Algorithm

The straightforward approach to pattern matching is, unfortunately, the least efficient. This algorithm basically consists of trying to match the pattern at every position in the text, halting each search as soon as a mismatch is found. If the prefix of the pattern appears frequently but not the entire pattern, then this algorithm demonstrates very bad behavior. Knuth-Morris-Pratt (1977) present an excellent example with the pattern a^ib being searched for in the text $a^{2i}b$. This algorithm must make $(i+1)^2$ comparisons. In general, the worst case running time is O(mn) where m is the length of the pattern and n is the length of the text to be searched.

This algorithm requires total random access, can only search for exact patterns, and has no preprocessing but *O(mn)* run-time. Several clever algorithms have been derived that have much better execution times.

3.5.2.2. Boyer-Moore

The Boyer-Moore (1977) algorithm takes a vastly different approach from all the other algorithms. Statements about this algorithm generally center around the average running time rather than the worst case. In one example, the software method could conceivably be better than any hardware method reviewed. Consider the pattern a^{100} (one hundred a's) and the text b^{1000} (one thousand b's). The Boyer-Moore algorithm would only make 10 comparisons to determine that the pattern does not exist anywhere in the text. Those 10 comparisons might invoke perhaps 7 machine instructions and two memory references for the tables but even at 90 memory references (10*(7+2)) the fact that the pattern does not occur is obtained much faster than the 1000 memory references that the best hardware algorithms would have had to make.

The Boyer-Moore algorithm accomplishes this by matching the pattern starting from the right end rather than the left as the other algorithms do. Finding a "b" in the *hundred*th position and knowing that a "b" does not appear anywhere in the pattern, allows the algorithm to avoid having to ever investigate the first 99 text characters.

This accounts for the algorithm at its best behavior. A worst case analysis is quite complicated but well documented in many publications (Boyer and Moore 1977; Galil 1979; Horspool 1980; Knuth, Morris, and Pratt 1977) which have analyzed and improved the algorithm. The worst case analysis yields a preprocessing complexity of *O(m)* and a run-time complexity of *O(n)* for an overall complexity of *O(m+n)*. On an average, it is expected to take less than *n* comparisons.

The order in which the data is accessed is quite different and thoroughly discussed with regard to its implications for paging situations. The preprocessing involves *O(m)* steps and tables for exact patterns only. The run-time requires *O(n)* steps.

3.5.2.3. Knuth-Morris-Pratt

While we related our algorithm back to language theory after the fact, Knuth, Morris, and Pratt (1977) independently derived a searching algorithm starting with language theory. Essentially a finite state automaton is created, but because the algorithm only searches for exact patterns, many assumptions can be made during the preprocessing.

This algorithm accesses the data sequentially with no backtracking. It searches for exact patterns only and has an *O(m)* preprocessing complexity for both time and table space and *O(n)* run-time.

3.5.2.4. Aho-Corasick

Aho and Corasick (1975) also recognized the usefulness of finite state automata for searching but determined a reluctance from programmers to use FSAs due to the complexity of programming the construction from a regular expression, especially if minimization was required. Like Knuth-Morris-Pratt, Aho-Corasick recognized the construction could be greatly simplified for restricted expressions. Their algorithm also searches for exact patterns but allows subpatterns and multiple patterns to be searched for as well.

The algorithm accesses the data sequentially with no backtracking, can handle multiple exact patterns, and has *O(m)* preprocessing in time and space with *O(n)* run-time.

3.5.2.5. Fischer-Paterson

Fischer and Paterson (1974) investigated extending the Knuth-Morris-Pratt algorithm to include fixed length don't care wild cards. They prove that the algorithm cannot be extended beyond its defined capabilities. They go on to present an algorithm that can recognize patterns containing embedded fixed length don't care characters but at the cost of losing linear run-time. Their run-time is $O(m \bullet (logn)^2 \bullet loglogn)$, which is less than the $O(mn)$ run-time of the traditional software search algorithm but greater than the $O(m+n)$ of our hardware algorithm.

3.5.2.6. Thompson

One of the earliest published software searching algorithms was written by Ken Thompson (1968). Thompson presents a method for performing a top-down, left derivation of a regular expression (Barrett and Couch 1979; Harrison 1978; Hopcroft and Ullman 1979; Salomaa 1969) driven by the data being searched. The basic concept is to create a list of all potential characters that can next be generated by the regular expression and then compare the next data character to this list.

The algorithm accesses data sequentially with no backtracking and allows full regular expression operators in the pattern but has prohibitively large complexities in preprocessing and run-time. The preprocessing involves three stages. First the regular expression is parsed for syntactic correctness and injected with an operator for concatenation. Next that regular expression is converted to a reverse-polish notation. Then the preprocessing generates assembly code to perform the regular expression derivation. The preprocessing can be accomplished in $O(m)$ time, but the coefficient for these three stages is quite large. The run-time has the potential to be exponential since a non-deterministic transition could cause a multitude of possibilities for the next derivation.

3.5.3. Hardware Searching Algorithms

The early hardware searching algorithms (Bird 1979; Haskin 1980; Roberts 1977) utilized various memory hardware techniques to improve searching performance. This concept works well for exact patterns but develops problems for more complex patterns. The exponential growth of the software algorithms for regular expressions occurs in these algorithms (Haskin 1980; Roberts 1977) as well.

The advent of Very Large Scale Integration (VLSI) (Mead and Conway 1980; Mukherjee 1986) allowed parallel algorithms to be considered fairly practical and generated some publications directly associated with parallel searching algorithms (Curry and Mukhopadhyay 1983; Foster and Kung 1980; Mukhopadhyay 1979). Since finite state automata (FSA) have been traditionally useful in computer language compilers and computer hardware design, algorithms (Foster and Kung 1981; Floyd and Ullman 1980; Trickey 1982) have been developed for automatic FSA layout in VLSI designs. While the primary purpose is not searching, these algorithms have a direct impact on this chapter.

3.5.3.1. Bird-Tu

A system for text retrieval was built using associative memory as its method to accomplish pattern matching (Bird 1979). In this system, associative memory addresses are the patterns. This type of search is very fast and allows simultaneous multiple pattern searches but restricts the pattern length to the size of a single address and restricts the types of patterns to exact word matching. The hardware can be expanded to allow more subpatterns but not to allow longer subpatterns.

The preprocessing only involves loading the patterns into the associative memory and the search looks for matches from the memory. It has multiple-exact-pattern capability preprocessed in $O(m)$ and run-time in $O(n)$.

3.5.3.2. Roberts

The Central Intelligence Agency (CIA) developed a hardware searching system based on Bird Indexing (Roberts 1977) to reduce memory overhead. Instead of an entire word being used for the state transition information, a single bit is used in conjunction with a base offset. This technique works quite well for exact patterns but gets complicated for more complex patterns. Roberts acknowledges this and provides modifications for *jump transitions* and *retry states* which are necessary when wild cards are introduced. Unfortunately, these modifications complicate the timing of data flow since the time to process a single character is variable. Furthermore, the exponential explosion of the software algorithms is present for this algorithm as well if full regular expressions are attempted.

3.5.3.3. Haskin

Haskin (1980) avoids the exponential explosion in states by keeping the regular expression non-deterministic. His hardware consists of having multiple modules each containing a copy of the NFSA. When a non-deterministic decision must be made, he has hardware to control which modules take which decision. While avoiding the exponential growth in the preprocessing, sufficient different modules must be present for all the non-deterministic decisions during the search which can be exponential. Furthermore, **each** module must have a memory that is $|\Sigma|$ wide by m tall.

3.5.3.4. Foster-Kung

The first searching algorithm published by Foster and Kung (1980) is nearly identical to the method designed by Mukhopadhyay (1979) with the same pattern complexity. However, in their system, the pattern is continually cycling through the chip rather than remaining in each cell. This movement of the pattern requires complicated timing which leaves half of the cells inactive at any given moment. Furthermore, their hardware performs a hierarchical comparison from the most significant bit to the least, which further complicates the timing by requiring the characters to be fed into the chip sequentially and the comparison to be pipelined.

This work was later expanded (Foster and Kung 1981) to compile regular languages into a hardware recognizer. Their technique adds two new types of cells for alternation and closure then hardwires a tree structure of the expression. They go on to show that the tree can be laid out in a structure that has length $O(m)$ horizontal cells by $O(\log m)$ height in routing.

3.5.3.5. Floyd-Ullman-Trickey

Floyd and Ullman (1980) and subsequently Trickey (1982) have pursued a Programmable Logic Array (PLA) approach to silicon compilation of regular expressions. A PLA can implement an NFSA by having outputs that feed back into the PLA as states. The combinations of the current input character and the current state are used to determine the next state. Since each state has its own line, non-determinism can be represented by multiple states active at once. They quantify the maximum number of states that might be produced during the generation of the NFSA as *2m* and the maximum number of arcs as *4m*. Since the states are both outputs and inputs, the size of the PLA is $|\Sigma| + 2(2m)$ wide by *4m* tall for $O(m^2)$ area.

Floyd and Ullman also discuss construction of a custom routed recognizer such as our solution shown in Figure 3.21 and quantify its dimensions to be $O(\sqrt{m})$ on a side.

3.5.4. Blocked Algorithms

A literature search of searching algorithms would certainly lead to the works of Galil, Seiferas, and Lee (1983; 1984; 1986). Their papers are appropriate and related but make the assumption that the data is blocked and the search is restricted to each block. They then go on to utilize a processor per data item rather than per pattern item. This makes them *O(n)* and inappropriate for comparison to the other algorithms discussed here which assume a fairly small *m* (length of the pattern) and an extremely large *n* (length of the data).

3.5.5. Curry-Mukherjee

The regular expression searching algorithm of section 3.2.4 is capable of searching for any patterns that can be specified by any of the algorithms described, but can be overkill for some applications. The area consumed for the communication bus could be used for more cells or other logic in the hardware design. Therefore, we will examine the algorithm comparison criteria for both the multiple-patterns-with-wild-cards algorithm of section 3.2.3 and the regular-expressions-with-wild-cards algorithm of section 3.2.4. These will be referred to as algorithm 1 and algorithm 2 respectively.

3.5.5.1. Order of Data Access and Run-Time Complexity

Both algorithms 1 and 2 access the data sequentially and require examination of each data item only once. As was proven with the n-MOS implementation, the algorithm can be designed to run fast enough to accept the data as fast as it can be read from memory. This means the run-time complexity is not only $O(n)$, but in reality $1n$.

The Foster-Kung and Floyd-Ullman-Trickey algorithms can conceivably make the same claims. All other software and hardware algorithms described require considerably more "cycles" per data item. Keeping the overhead to less than one order of magnitude larger ($10n$) would be quite difficult.

The Boyer-Moore algorithm is the only algorithm described which does not have to examine every data item at least once. Unfortunately, it requires complete random access of the data, making it unusable for applications which stream the data sequentially such as back-end architectures for text retrieval. The "intuitive" algorithm is the only other algorithm requiring random access.

3.5.5.2. Pattern and Preprocessing Complexities

It has been strongly emphasized throughout this chapter that any algorithms that require conversion from a non-deterministic representation to a deterministic one can experience exponential explosion in the number of states and the preprocessing effort. The intuitive, Boyer-Moore, Knuth-Morris-Pratt, Aho-Corasick, and Bird-Tu algorithms all restrict the pattern to *exact patterns* in order to guarantee a deterministic result in linear time. The addition of wild cards is sufficient to force the software algorithms out of linearity and parenthesized subexpressions push the hardware algorithms out of linearity.

Algorithm 1 proves that multiple patterns with wild cards can be implemented in hardware and preprocessed in linear time. The first Foster-Kung algorithm achieves the same result.

Algorithm 2 incorporates all of the regular expression operators, in addition to wild cards, and maintains a linear preprocessing but requires quadratic space. The Foster-Kung and Floyd-Ullman-Trickey algorithms also accept full regular expressions with linear preprocessing and also have polynomial space requirements. Neither addresses wild cards because they are not implementing the regular expression necessarily for searching, but both could address wild cards with little effort.

The Foster-Kung regular expression algorithm requires *O(m)* cells and *O(log m)* routing for *O(m log m)* space requirements. The Floyd-Ullman-Trickey PLA technique requires *O(m)* states for width and *O(m)* transitions for height resulting in *O(m^2)* space requirements. Algorithm 2 is equivalent in space complexity to the Floyd-Ullman-Trickey algorithm but larger than the Foster-Kung complexity. However, algorithm 2 has the advantage of being reprogrammable for a new expression and both of the other techniques are hardwired.

Floyd and Ullman also show that a hardwired version very similar to algorithm 2 could be implemented with $O(\sqrt{m})$ width and $O(\sqrt{m})$ height yielding linear space requirements. If reprogrammability is not a requirement, then constructs similar to those shown in Figure 3.21 can be used to hardwire a version of algorithm 2 in *O(m)* space.

3.5.5.3. Data Element Size

At first glance it would seem that once algorithms 1 and 2 were implemented for a particular data size, it is a fixed quantity. A well structured software program might be able to change a character declaration to an integer declaration, recompile, and suddenly the algorithm searches for four-byte data items instead of one-byte items. What can be done about the hardware algorithms?

It turns out that algorithms 1 and 2 can both be utilized for varying data sizes after implementation. Let us presume the algorithms are implemented with the *Pattern Latch* and *Comparator* sizes set to 8-bits (one byte). If the data elements are smaller than a byte, then padding the data elements with zeros to fill in a full byte is the trivial solution. If the data elements are larger than a byte, then the concatenation operation can be used to resolve the latch overflow. Consider prepending a 1 onto the most significant bit of the start of every new data item and prepending a 0 for every subsequent 7 bits of data for each data item. Then the concatenation of adjacent hardware cells enables the match of a single data item. If the *fldc* and *vldc* are set appropriately in the same number of adjacent cells, then wild cards are properly implemented. The routing-cell definition of algorithm 2 is not affected since its only purpose is to route state information to other cells.

Therefore, one of the major obstacles to a hardware searching algorithm is resolved by the nature of concatenation. This same solution could conceivably be applied to all of the software and hardware algorithms described in this chapter since concatenation is the single operation that is constant throughout all searching algorithms.

3.6. Conclusions

Searching is one of the most prevalent string operations and the focus of considerable attention in the computing industry. Moreover, searching is not limited to just strings.

Uniprocessor searching techniques suffer extremely poor time and space complexities during the preprocessing of complex patterns and are at least an order of magnitude slower than multiprocessor techniques during the run-time phase. Previous publications have demonstrated that linear time and polynomial space multiprocessor algorithms for regular expressions are realizable and have applied the solutions to silicon compilers. Our algorithm directly applies regular expressions to the problem of searching, and provides a fully reprogrammable solution in like time and space requirements.

While searching was the primary emphasis in the presentation of the algorithm, it is conceivable that the technique could be applied to any problem which can be solved by a regular expression, finite state automaton, or regular grammar.

The *String Coprocessor* discussed in the rest of the dissertation will incorporate these searching algorithms into several string operations.

CHAPTER 4

THE STRING COPROCESSOR

The hardware searching algorithm of the previous chapter requires external logic to feed it the patterns and the data. This is basically accomplished through two loops. As frequently mentioned in the early chapters, all string operations are built around loops since the size of any string is arbitrary. This chapter combines the regular expression hardware searching algorithm of Chapter 3 with additional logic to implement the character string operations defined in Chapter 2.

4.1. The System Interface

Most central processing unit (CPU) manufacturers and computer vendors have options for the addition of a floating point coprocessor. The floating point coprocessor communication is generally proprietary and tightly coupled to the specific CPU implementation. The *String Coprocessor* design will attempt to be generically applicable to any computer system independent of the CPU and the system bus design. The only assumptions are:

(1) The coprocessor is accessible by the CPU.

(2) The memory is addressable on single character boundaries.

(3) The coprocessor is allowed to perform direct memory access (DMA).

The *String Coprocessor* contains some registers which are loaded by the CPU in preparation for the operation. This preparation generally consists of loading the addresses of the strings to be manipulated and then loading the operation into the instruction register of the coprocessor. Once the instruction register of the coprocessor is set, the coprocessor enters DMA mode and performs the string operations directly on the data. Upon completion of the operation, a status register is set in the coprocessor and control is passed back to the CPU. The CPU can then read the coprocessor status register to set its own condition codes appropriately.

In order for the CPU to access the coprocessor registers, an address enable input pad is combined with a read/write input pad, some register address input pads, and data input/output pads. The size of the registers depends on the addressability of the CPU. Likewise, the number of data input/output pads depends on the data path size of the system. These are not necessarily absolute values but simplify the description of the design. A worst case scenario might imagine a single data pad used to serially feed the data in or out a single bit at a time and the address of the string operands in the registers to be limited and added to a base offset. These compromises and possibly more could be considered but are not necessary given the already advanced levels of semiconductor technology. The input/output pad counts and transistor densities of current technology allow a complete implementation of the *String Coprocessor* having separate data and address pads with total parallel transfer.

The chip will perform DMA by placing an address on the address pads and then requesting a read or write. If the operation is a read, the coprocessor will wait for a signal indicating the requested information is on the data pads. If the request is for a write, the coprocessor will hold the data on its data pads until it receives a signal indicating the write is complete. This technique allows external logic to perform the specific protocols of the bus handshaking while allowing the coprocessor to remain generic.

4.2. The String Coprocessor Registers

The string operations defined in Chapter 2 have at most three operands. At least three registers are therefore required, each of sufficient size to hold a full address. These will be called the *String Registers* and labeled S1, S2, and S3. Each of these registers can be gated to the address bus and to or from the data bus. Additionally, they can be cleared, incremented by one, decremented by one, and always provide an output indicating if the content of the register is zero or not.

In addition to those three registers, some other registers are necessary. One is to hold the termination character which delimits the end of a character string (labeled TERM), another to hold the status of the coprocessor and its operation (labeled SR for Status Register), and another to hold the current operation (labeled IR for Instruction Register). The control lines for the additional registers include clearing and incrementing the instruction register and setting the status register. The number of bits required for these additional three registers is quite small and, presuming a reasonably large address space, will likely total to less than one of the three string registers. These additional registers will be combined together as subregisters and be configured as a fourth string register.

The last two registers required for the *String Coprocessor* are used to hold the current characters of the strings during an operation. The contents of these two registers (labeled C1 and C2) are continuously compared against each other and the terminator character in TERM for the results of C1 = TERM, C2 = TERM, C1 = C2, and C1 > C2. These four results are always made available to the control section of the chip. The logic which performs this comparison is labeled CLU for Character Logic Unit.

The addresses for these registers will be:

TABLE 4.1. Coprocessor Register Addresses

Address	Register
0	S1
1	S2
2	S3
3	IR
4	SR
5	TERM
6	C1
7	C2

4.3. The Character Logic Unit (CLU)

The combination of the registers, CLU, regular expression searching algorithm, and a control section provide all of the hardware necessary for the implementation of the *String Coprocessor*. The load/store, increment, decrement, and clear requirements of the string registers are quite common operations for registers and require no special attention. The requirements of the four outputs of the CLU, on the other hand, should be investigated to insure that no critical path is introduced in the *String Coprocessor*, making it unnecessarily slow. As was done with the searching algorithm, an n_MOS implementation project was undertaken (Crystal and Hendry 1983) to make the best use of semiconductor technology and confirm no nasty realities exist with the paper design. As might be expected, pass transistors are used in addition to traditional combinational logic to reduce the gate delay of the critical path. The following diagram shows the floorplan of the CLU.

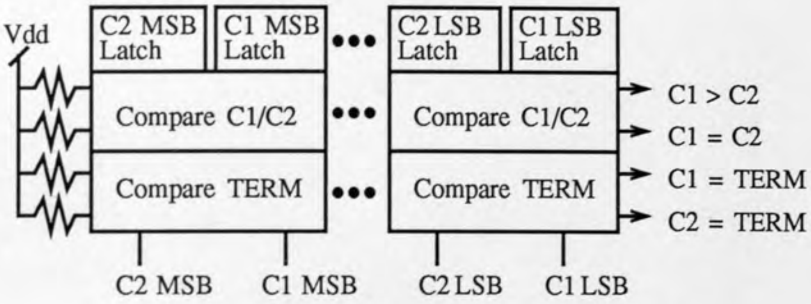


Figure 4.1. Character Logic Unit

The latches used to hold the bits of each character are identical in design to the latches of Chapter 3. The most significant bit (MSB) is placed in the left-most latch and the least significant bit (LSB) in the right-most. Likewise, the *simultaneous exclusive-nor* logic used to compare the *Pattern Latch* against the current data element in the searching algorithm can be applied to the C2 = TERM and C1 = TERM lines. The C1 > C2 logic is new and affects the C1 = C2 logic as shown in the following diagram.

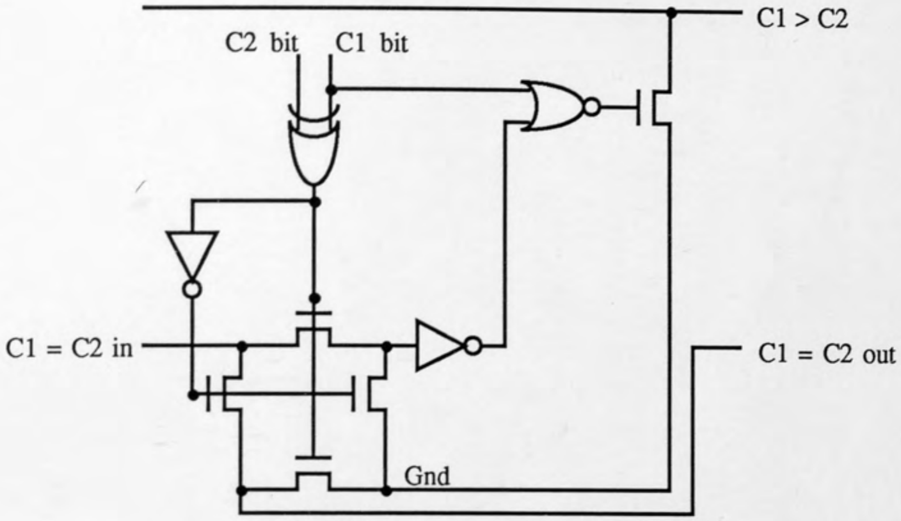


Figure 4.2. Compare C1 and C2

This logic is designed such that the C1 > C2 line can be grounded only if all more significant bits were equal **and** C2 is a 1 **and** C1 is a 0 for this cell. All other combinations leave the C1 > C2 line untouched. The following figure shows the compare logic as it was implemented in n-MOS.

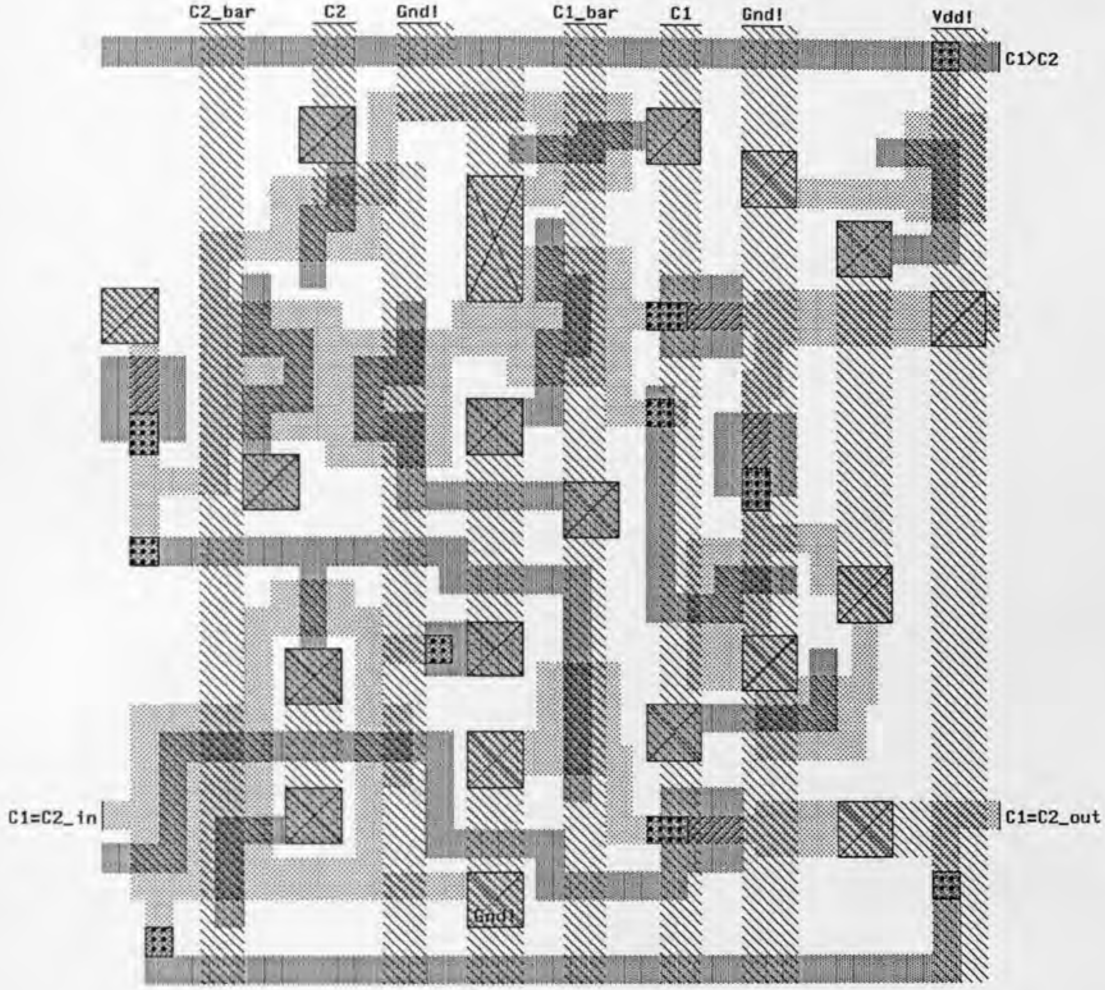


Figure 4.3. n-MOS Implementation of the Compare Cell

4.4. Searching Hardware

The searching requirements for the operations detailed in Appendix A are extremely simple compared to the power of the searching algorithms presented in Chapter 3. All of the operations are based on movement of pointers which in turn are based on the current character's membership to a set of characters.

The trivial design of a one-bit memory with the number of words equal to the size of the alphabet would be sufficient to accomplish these searching operations. The preprocessing would clear all words to zero then assign a one to each word whose address is the ordinal value of the character in the search set. Reading the memory during the search will indicate membership or lack of membership in the search set for the current character.

The multiple-exact-pattern algorithm of Chapter 3 can also perform this search by considering each character of the search set an individual pattern and setting the End Of Pattern (EOP) bit for each character. Likewise, considering the search set to be the alternation of several single character patterns allows the regular expression algorithm to perform the search.

The control unit of the coprocessor will be implemented as if the multiple-wild-card-pattern algorithm of Chapter 3 is incorporated as the searching hardware. In doing this, we show how the control unit can perform both the preprocessing and run-time phases of the searching operations using our hardware searching algorithm.

4.5. Component Communication

All of the logic is now defined with the exception of the control section to implement the operations. The following block diagram defines the interconnections of each of the sections of logic previously defined.

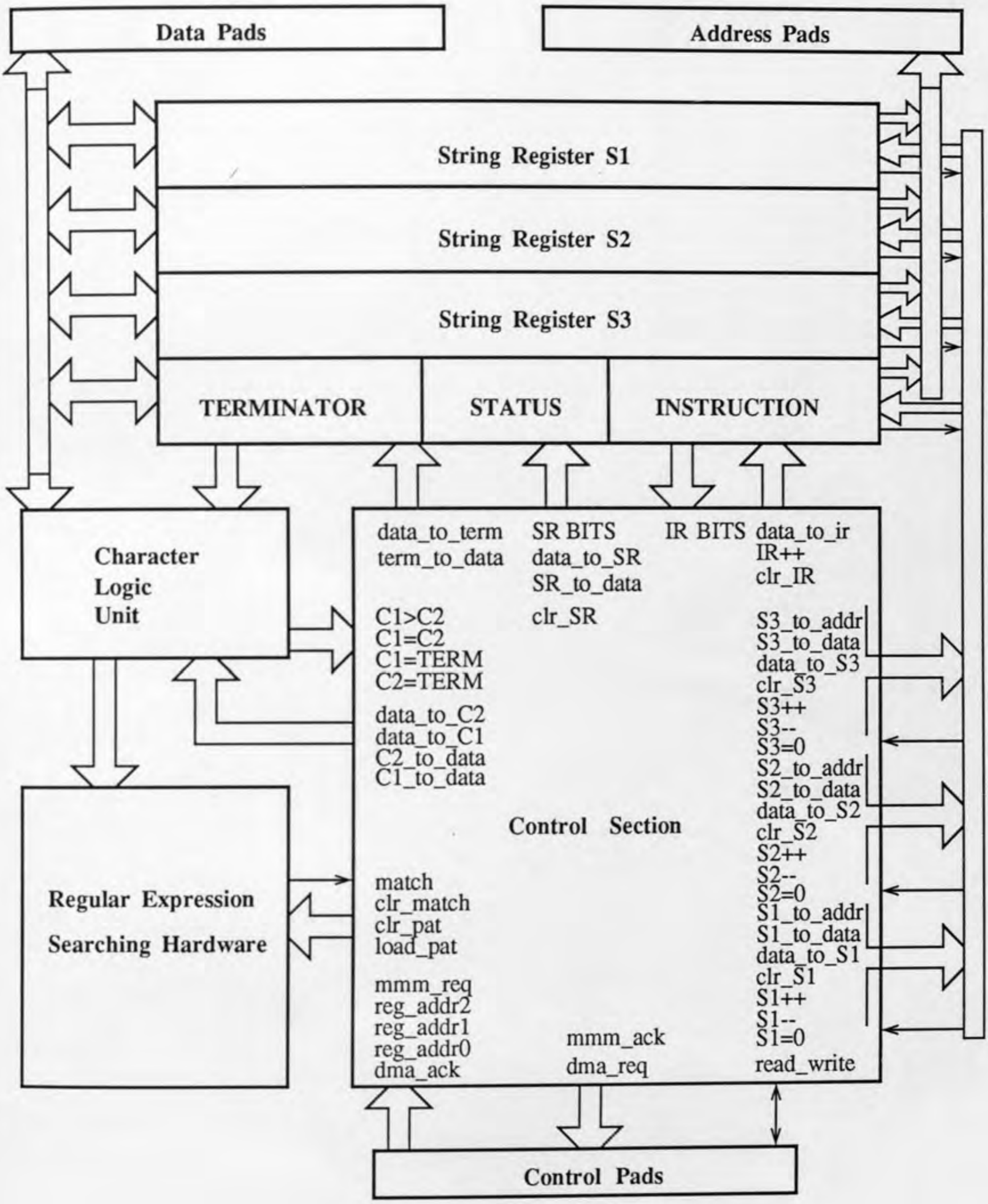


Figure 4.4. String Coprocessor Block Diagram

As discussed in the *System Interface* section, the *data bus* and *address bus* are completely separate and parallel structures. All of the registers are accessible to the external world and each other bi-directionally through the *data bus*. This requires the data pads to have three states (called a tri-state pad). These states are:

(1) Input - when loading the bus from the outside world.

(2) Output - when providing data to the outside world.

(3) High Impedance - when no coprocessor data communication is occurring with the outside world. This prevents the coprocessor from affecting any other circuitry.

The address pads are only used when the coprocessor needs to read data from memory during direct memory access (DMA) mode during an operation execution. These pads need to have an output mode when in use and a high impedance mode when not in use.

The *mmm_enable* (memory mapped mode enable) input pad is used when the external world is accessing the coprocessor's registers. The *read_write* tri-state pad is enabled for input by the *mmm_enable*, and the three *reg_addr* (register address) input pads indicate which register to read or write. The *mmm_ack* pad is used as a "memory mapped mode acknowledgment" informing the outside world that the coprocessor has completed the read/write.

That same *read_write* tri-state pad can be used by the coprocessor when in DMA mode. When the coprocessor wants to read or write to memory, it takes the following steps.

READ

(1) Turn on the *dma_req* pad indicating a memory transfer is requested and simultaneously set the *address_pads* and the *read/write* pad.

(2) Wait for the *dma_ack* pad to be set indicating the data is on the *data pads*. If the *dma_ack* is set, then bring in the data off of the *data_pads*, release the *dma_req* pad, and continue execution of the operation. If the coprocessor times out waiting for *dma_ack*, then release the *dma_req* pad, set the *status* register, and halt execution of the operation.

WRITE

(1) Turn on the *dma_req* pad, set the *data pads*, *address pads*, and *read/write* appropriately.

(2) Wait for the *dma_ack* pad. If set, then release the *dma_req* pad and continue. If the coprocessor times out waiting for *dma_ack*, then release the *dma_req* pad, set the status register, and halt execution of the operation.

The remainder of the control lines are fairly self explanatory. The abbreviation "clr" is used for "clear", "SR" for "status register", and "IR" for "instruction register". A "++" or "--" after a register name denotes increment or decrement by one respectively. All control lines can be active in parallel allowing such things as *S1++*, *S2++*, *S3--*, and *data_to_C1* to occur simultaneously.

The *control section* then takes all of its inputs and sets the control lines appropriately for a given request.

4.6. Control Section

The *Control Section* can be implemented in a variety of ways. Perhaps the simplest means is through a finite state automaton implemented through a Programmable Logic Array (PLA). Another approach might be to micro-program the control section using a micro-program counter, micro-instruction decoder, and a micro-program control store.

The actual design details are not as important as the control line definitions and the micro-programs of the string operations. The control lines are listed in the previous section in the block diagram of the coprocessor. The micro-programs for each string operation will be given in this section.

In order to simplify and significantly shorten the definition of each micro-program, a Register Transfer Language (RTL) definition is used. Furthermore, an assumption is made that some micro-program subroutines can be utilized and shared among all the micro-programs.

4.6.1. MEMORY

The operations defined by MEMORY are all based on a length parameter rather than the terminator character. Each MEMORY operation and its micro-program is presented in this section.

4.6.1.1. MEMCCPY

The *memccpy(s1,s2,c,n)* operation copies characters from *s2* to *s1* until the character *c* has been copied or *n* characters have been copied. In preparation for this operation, the CPU will load the address of *s1* and *s2* into the coprocessor registers S1 and S2, respectively. The character *c* is loaded into C2 and the length *n* is loaded into S3. Finally, the operation code is loaded into the coprocessor IR. The following micro-program is then executed.

```
memccpy        if (s3=0) clr_s1, goto done; else s2_to_addr, gosub read_into_c1
               s1_to_addr, gosub write_c1
               if (c1 = c2) goto done; else s1++, s2++, s3--, goto memccpy
```

This operation leaves S1 pointing to the position of the copied *c* character or clears S1 if execution halted due to length rather than having copied *c*. Each line of the micro-programs represents one cycle of the coprocessor execution. In the first line of *memccpy*, a test is made to see if the length counter is exhausted. If so, S1 is cleared and execution is halted. If not, then the address of the next character in S2 to be copied is placed on the address pads and a micro-program subroutine to perform DMA is called to read that character into C1. The second line writes that value to the address pointed to by S1. The last line tests if the written character was the halt character. If so, then halt execution. Otherwise, increment the string pointers, decrement the counter, and loop through again.

4.6.1.2. MEMCHR

The *memchr(s,c,n)* operation searches for the first occurrence of *c* in the first *n* characters of *s*. If *c* is not found, a null value is returned. The micro-program expects *s* to be placed in S1, *c* to be placed in C2, and *n* to be placed in S3 prior to the start of the operation.

```
memchr         if (s3=0) clr_s1, goto done; else s1_to_addr, gosub read_into_c1
               if (c1 = c2) goto done; else s1++, s3--, goto memchr
```

4.6.1.3. MEMCMP

The *memcmp(s1,s2,n)* operation lexicographically compares the first *n* characters of *s1* and *s2*. The micro-program expects *s1*, *s2*, and *n* to be loaded into S1, S2, and S3, respectively. The result of the comparison is placed in the status register (SR).

```
memcmp      if (s3=0) then sr=0, load_sr, goto done; else s1_to_addr, gosub read_into_c1
            s2_to_addr, gosub read_into_c2
            if (c1>c2) then sr=1, load_sr, goto done
                  else if (c1<c2) then sr=-1, load_sr, goto done
                  else s1++, s2++, s3--, goto memcmp
```

If the entire length has been compared without finding a difference, then the strings are equal and SR is set to zero. Otherwise, a character is read from each string and compared. The last three lines of the micro-program are all executed simultaneously since the CLU provides all of the C1, C2 comparison results to the control section.

4.6.1.4. MEMCPY

The *memcpy(s1,s2,n)* operation is a block transfer of *n* characters from *s2* to *s1*. Once again *s1*, *s2*, and *n* are expected to be placed in S1, S2, and S3, respectively.

```
memcpy      if (s3=0) goto done; else s2_to_addr, gosub read_into_c1
            s1_to_addr, gosub write_c1
            s1++, s2++, s3--, goto memcpy
```

4.6.1.5. MEMSET

The *memset(s,c,n)* operation sets *n* characters to the value of *c* starting at *s*. The micro-program expects *s*, *c*, and *n* to be in S1, C1, and S3, respectively.

```
memset      if (s3=0) goto done; else s1_to_addr, gosub write_c1
            s1++, s3--, goto memset
```

4.6.2. STRING

These operations differ from those associated with MEMORY in that a terminator character is used to halt executions. Some operations may take a length operand but the terminator character always takes precedence over the length. These operations and their micro-programs are detailed in this section.

4.6.2.1. STRCAT and STRCPY

The *strcat(s1,s2)* operation concatenates a copy of string2 onto the end of string1. The *strcpy(s1,s2)* operation copies string2 into string1. The following micro-programs implement these two operations.

```
strcat          s1_to_addr, gosub read_into_c1
                if (c1 != term) s1++, goto strcat

strcpy          s2_to_addr, gosub read_into_c1
                s1_to_addr, gosub write_c1
                if (c1 = term) goto done; else s1++, s2++, goto strcpy
```

The *strcat* operation works by first finding the end of *s1* and then falling into the *strcpy* operation. The second line examines the CLU outputs to see if we have reached the end of *s1* or not. If not, then increment S1 and go back up to read the next character. If so, then fall into the *strcpy* routine to complete the operation. The *strcpy* operation reads each character of *s2* into C2 and writes it to *s1*. If the character just written to *s1* was the terminator, then the operation is done. Otherwise, increment *s1* and *s2* and continue the loop.

4.6.2.2. STRNCAT

The *strncat(s1,s2,n)* operation is similar to the *strcat* operation but concatenates at most *n* characters where *n* is placed in S3. The same code can be used for finding the end of *s1* but this operation cannot just simply fall into *strcpy*. Nor can it fall into the *strncpy* operation since *strncpy* will pad to use **all** of *n* characters rather than **at most** *n* characters. The micro-program is shown here.

```
strncat         s1_to_addr, gosub read_into_c1
                if (c1 != term) s1++, goto strncat
strncat1        if (s3 = 0) goto write_term; else s2_to_addr, gosub read_into_c1
                s1_to_addr, gosub write_c1
                if (c1 = term) goto done; else s1++, s2++, s3--, goto strncat1
```

The first two lines are identical to *strcat*. The rest of the code is similar to *strcpy* with the exception that the coprocessor register S3 is used as a counter to know when *n* characters have been copied. If the value of S3 becomes zero before the end of *s2* is detected, then force a terminator character to be written to *s1* and quit. Otherwise, read the next character from *s2* and write it to *s1*.

4.6.2.3. STRNCPY

The *strncpy(s1,s2,n)* operation copies exactly *n* characters, from *s2* to *s1*. The *s1* result will not have a terminator written out if the *length* of *s2* is greater than *n*. *s1* will be null padded if the length of *s2* is less than n. The micro-program for *strncpy* is:

```
strncpy         if (s3 = 0) goto done; else s2_to_addr, gosub read_into_c1
                s1_to_addr, gosub write_c1
                if (c1 != term) s1++, s2++, s3--, goto strncpy
strncpy1        if (s3 != 0) term_to_data, s1_to_addr, gosub write; else goto done
                s3--, goto strncpy1
```

The first three lines copy *s2* as long as *s2* has characters and *n* is not exhausted. The second loop pads the string with terminators if *len(s2) < n*.

4.6.2.4. STRCMP and STRNCMP

These operations lexically compare two strings to each other returning the difference. The CLU allows the SR to be set appropriately but does not provide the exact difference of two characters. This should be sufficient for proper execution of this operation. However, if the actual difference is required, then C1 and C2 hold the first characters that were different and the CPU can read them. The mico-program for *strcmp* is:

```
strcmp          s1_to_addr, gosub read_into_c1
                s2_to_addr, gosub read_into_c2
                if (c1 = term & c2 = term) SR= EQ, load_sr, goto done;
                        else if (c1 = term & c2 !=term) SR=LT, load_SR, goto done;
                        else if (c1 != term & c2 = term) SR=GT, load_SR, goto done;
                        else if (c1 != term & c2 != term & c1 > c2) SR=GT, load_SR, goto done;
                        else if (c1 != term & c2 != term & c1 != c2 & c1 !> c2) SR=LT,
                                load_SR, goto done;
                        else s1++, s2++, goto strcmp
```

This is actually a three line micro-program but spread out for readability. The *strncmp(s1,s2,n)* operation is identical except the first line becomes:

```
strncmp         if (s3 = 0) SR=EQ, load_sr, goto done; else s1_to_addr, gosub read_into_c1
```

and the last line decrements S3 at the same time as S1 and S2 are incremented.

4.6.2.5. STRLEN

This operation returns the length of a string not including the terminator. Its micro-program is quite simple.

```
strlen          clear_s3
strlen1         s1_to_addr, gosub read_into_c1
                if (c1 = term) goto done; else s1++, s3++, goto strlen1
```

4.6.2.6. STRCHR, STRRCHR, INDEX, and RINDEX

strchr(s,c) finds the first occurrence of the character *c* in the string *s* and *strrchr* finds the last occurrence. The old names for *strchr* and *strrchr* are *index* and *rindex*, respectively. A zero is returned if *c* does not appear in *s*. The address of *s* is placed in S1 and *c* in C2.

```
strchr          s1_to_addr, gosub read_into_c1
                if (c1 = c2) then goto done, if (c1 = term) clr_s1, goto done; else s1++, goto strchr

strrchr         clr_s3
strrchr1        s1_to_addr, gosub read_into_c1
                if (c1=c2) s3=s1
                if (c1!=term) s1++, goto strrch1; else goto done
```

strchr goes right through *s* looking for *c*. Each time *strrchr* encounters *c*, it copies the pointer into S3. When the terminator is encountered, S3 will be left as zero if *c* was never found. Otherwise, S3 will be pointing to the last occurrence of *c*.

4.6.2.7. STRPBRK and STRCSPN

The *strpbrk(s1,s2)* and *strcspn(s1,s2)* operations look for the first occurrence of a character from *s2* in *s1*. This is in effect a *scan* operation. The two operations return different values. *strpbrk* returns a pointer to the first occurrence or zero if there is none. *strcspn* returns the length before an occurrence was found. These operations can be accomplished by loading *s2* into the searching logic, setting the *EOP Latch* for every character and then performing a search on *s1*.

```
strpbrk         SR=1, load_SR, clr_s3, clr_pat
                SR_to_data, data_to_c2
strpbrk1        s2_to_addr, gosub read_into_c1
                if  (c1!=term) load_pat, s2++, goto strpbrk1
                clr_match
strpbrk2        s1_to_addr, gosub read_into_c1
                if (c1=term) then clr_s1, goto done;
                        else if (match=0) s1++, s3++, goto strpbrk2;
                        else if (match=1) goto done;
```

The status register is the only register into which the coprocessor can place a value. It is used to set C2 so that the pattern will have *EOP* set for each character. The subsequent read or done subroutines will reset the SR appropriately.

This same micro-program accomplishes both operations since S3 could be used as a counter during the search with no additional overhead. For *strpbrk*, the CPU should return the value in S1, and for *strcspn*, the value in S3.

4.6.2.8. STRSPN

The previous section performed a *scan* and this section performs a *span*. The same technique is used except we continue the operation **as long as** we match instead of **until** we match.

```
strspn          SR=1, load_SR, clr_s3, clr_pat
                SR_to_data, data_to_c2
strspn1         s2_to_addr, gosub read_into_c1
                if (c1!=term) load_pat, s2++, goto strspn1
                clr_match
strspn2         s1_to_addr, gosub read_into_c1
                if (c1=term | match=0) goto done; else s1++, s3++, goto strspn2
```

4.6.2.9. STRTOK

The *strtok(s1,s2)* operation is fairly complex in its definition. In the first call to *strtok*, *s1* is pointing to the start of a string with text tokens separated by the delimiters specified in *s2*. The operation involves spanning any delimiters that may be on the front of the string and then scanning for the delimiters to mark off the text token. *s1* is left pointing to the first character of the text token or set to zero if no text token was found. The first delimiter after the text token is converted to a terminator character. Subsequent calls to *strtok* set *s1* to zero, indicating a continuation from where the last *strtok* left off. The subsequent calls can redefine the delimiters each time in *s2*.

This operation could be implemented by sequential calls to the *strspn* (span) and *strcspn* (scan) operations already implemented in the coprocessor. But in keeping with the concept of micro-programming as much of each of the operations as possible, here we present a separate micro-program to accomplish *strtok*.

```
strtok          SR=EOP, load_sr, clr_pat
                sr_to_data, data_to_c2
strtok1         s2_to_addr, gosub read_into_c1
                if (c1 != term) load_pat, s2++, goto strtok1
                clr_match, s1_to_data, data_to_s3
strtok2         s1_to_addr, gosub read_into_c1
                if (c1 = term) clr_s1, goto done;
                        else if (match = 1) s1++, s3++, goto strtok2
                s3++, clr_match
strtok3         s3_to_addr, gosub read_into_c1
                if (c1 = term) s3++, goto done;
                        else if (match = 1) goto write_term;
                        else s3++, goto strtok3;
```

This micro-program presumes each call to *strtok* is a first call. It copies S1 into S3, spans the delimiters incrementing both S1 and S3, leaves S1 at the start of the token, scans for delimiters incrementing S3, and writes over the first delimiter found. The micro-program could be written to check if S1 = 0 and copy S3 to S1 assuming no other string operation has been performed since the last *strtok*, but that seems an extremely unlikely situation. Instead, we leave it up to the *strtok* calling routine to keep the new starting address for the hardware and load the registers each time as if this were a first call to *strtok*.

4.7. Conclusions

The hardware design of the coprocessor is capable of implementing all of the desired operations with fairly short and simple micro-programs. This chapter has presented that design and those micro-programs. However, simply having the micro-programs defined does not give any quantification of the improvement that might be realized through use of the coprocessor. The next chapter presents a simulation of the coprocessor and compares the performance of the operations against software implementations.

CHAPTER 5

PERFORMANCE IMPROVEMENT

The micro-programs in the previous chapter illustrate the simplicity of implementing non-numeric operations in the coprocessor hardware design. The VLSI implementations of the searching algorithm and character logic unit prove the viability of realization. This chapter quantifies the levels of improved performance that might be expected if the coprocessor were incorporated into a system. The quantification is accomplished by simulating the coprocessor in a system and applying a series of programs to the simulation. Each program is run once as if the coprocessor were **not** there and run again as if the coprocessor **were** there. The difference in the two runs quantifies the improvement attributable to the coprocessor. Several other UNIX filter programs may be run through the simulator as well in order to compare the performance of the coprocessor runs.

Two levels of improvement will be quantified. The improvement of individual operations is examined first. Then the operations are incorporated into a series of programs representative of the types of tasks string oriented languages might need to perform. Incorporating the operations into programs quantifies the overhead of loading the coprocessor with the operands and retrieving the value after completion of the operation.

5.1. Simulation Environment

As explained in Chapter 2, our selection of operations is based on the string operation library of the UNIX operating system. In recent years, a number of different computer vendors have utilized the Motorola MC68020 Central Processing Unit (CPU) as the processor for their UNIX systems. This multi-vendor acceptance of the MC68020 as a UNIX CPU combined with the complex instruction set computer (CISC) nature of the MC68020 architecture makes it appropriate as a baseline for determining the magnitude of performance improvement that might be realized by augmenting a UNIX system with our string coprocessor.

The specific computer system on which the simulations were run is a Sun Microsystems Sun-3 system. The system has an MC68020 CPU running the SunOS 3.5 version of the UNIX operating system.

The MC68020 contains an instruction cache, and utilizes instruction prefetch and instruction execution overlap techniques to improve performance. These factors make exact timing calculations extremely difficult. Instruction timings are published in cycle counts (Motorola 1985). One cycle equals one tick of the system clock. For a given instruction, three cycle counts are listed.

(1) Best Case - the time the instruction would take if it is already in the instruction cache and has maximum overlap with other instructions.

(2) Cache Case - when the instruction is in cache but has no overlap.

(3) Worst Case - when the instruction is not in cache and there is no overlap.

The *best case* and *cache case* numbers are dependent on instruction sequences and context conditions and are therefore unpredictable from one run of a program to the next. Hence, the *worst case* number is the only consistent measure of cycles for a particular program run. All cycle counts in this chapter are based on the *worst case* timing and assume zero wait state memory. The coprocessor is assumed to have the same system clock and assumed to take equivalent time to read or write memory locations.

5.2. Individual Operation Improvement

The nature of all of the operations is to have some setup effort, followed by the loop performing the operation, followed by returning the result. The setup and result stages are constant for each operation regardless of the size of the operands. The loop (or multiple loops) of an operation determine the complexity. We have stated that the copying and comparing operations will show a linear improvement and that the searching operations will show a complexity improvement. The magnitude of those improvements for the looping portions of the operations is shown in the following table.

TABLE 5.1. Individual Operation Improvement

	Operation	Software Cycles	Coprocessor Cycles	Improvement
1	memccpy	46n	7n	6.57
2	memchr	28n	4n	7.00
3	memcmp	53n	7n	7.57
4	memcpy	18n	7n	2.57
5	memset	14n	4n	3.50
6	strcat	19m+18n	4m+7n	4.75,2.57
7	strncat	19m+18n	4m+7n	4.75,2.57
8	strcmp	42s	7s	6.00
9	strncmp	50s	7s	7.14
10	strcpy	18n	7n	2.57
11	strncpy	18n	7n	2.57
12	strlen	19m	4m	4.75
13	strchr	35m	4m	8.75
14	strrchr	35m	5m	7.00
15	strpbrk	(37m+69)n	4m+4n	poly to linear
16	strspn	(47m+49)n	4m+4n	"
17	strcspn	(47m+49)n	4m+4n	"
18	strtok	(84m+118)n	8m+8n	"

m = len(s) for single string operations or len(s1) for two string operations
n = len(s2) or the length operand, whichever is shorter
s = the shorter of len(s1) or len(s2)

The linear improvements range from 2.57 to 8.75 times the performance of the software operations. In examining the software implementations, it was apparent that the optimizer had made efficient use of two powerful instructions of the MC68020. These are the Decrement-Branch-Condition-Code (DBcc) and MOVE instructions. DBcc first checks the current condition code in the status register, and if the appropriate condition is true, it falls through to the next instruction. If the condition was false, then a counter is decremented and compared to -1. If the counter is -1, it falls through to the next instruction. Otherwise, the branch is taken.

Through proper selection of the condition code and counter, DBcc can be efficiently used for operations which are halted through a counter or a comparison. The MOVE instruction is capable of performing a memory-to-memory transfer and automatically incrementing two pointer registers. A tight loop that is closely comparable to the coprocessor loop can be implemented by combining the functionality of the DBcc and MOVE instructions. The 2.57 fold improvement therefore becomes representative of the improvement due to the loop being contained in the coprocessor control section rather than in memory.

The larger linear improvements add more parallelism and utilize the character logic unit (CLU) to their advantage. The real improvement is represented in the searching operations. The software implementation utilizes the "intuitive" algorithm described in Chapter 3. Our coprocessor utilizes the exact-multiple-pattern hardware searching algorithm as described in Chapter 4. Not only is there a basic complexity improvement from $O(mn)$ to $O(m+n)$, but the coefficients are also quite small.

Knowing the improvement of the individual operations does not quantify the usefulness of the coprocessor. If an operation is dramatically improved but only accounts for a small percentage of execution time, then the overall performance of the execution is not greatly affected. The following sections quantify the improvement that might be expected for a string language at the program level.

5.3. Simulation Technique

The task of simulating an entire system, including file input/output, is somewhat overwhelming. Even a microprocessor like the Motorola MC68020 has over 100 instructions, most of which can use 18 different addressing modes. The technique we decided to employ makes use of the UNIX PTRACE function to control the normal execution of a program.

PTRACE allows a task to completely control a child task. This control extends to setting break points, reading/writing registers/memory, as well as many other features. Most importantly, it allows control to single step execution one machine instruction at a time. PTRACE is generally used by debuggers, but in our case, provides a means to have an entire program under our control. This is accomplished by invoking the simulator and passing the command line of the program to be executed as parameters. For example, the UNIX command *wc* provides the line, word, and byte count of information in a file. The simulator is called *collect* and gathers statistics about the execution of the program. The following is an invocation of the simulator on *wc* and its results.

```
% collect wc ~/.login

     23    100    651 /usr/tim/.login

Collect Statistics:
CPU Cycles =            137105
Coprocessor Cycles =    0
Total Cycle Count =     137105

Cycles per routine sorted by time:
  1   Routine = _main               Cycles = 105692  % = 77.088363
  2   Routine = __doprnt            Cycles = 12924   % = 9.426352
  3   Routine = _memchr             Cycles = 3044    % = 2.220196
  4   Routine = _cfree              Cycles = 2279    % = 1.662230
  5   Routine = __fwalk             Cycles = 1892    % = 1.379964
  6   Routine = _fclose             Cycles = 1404    % = 1.024033
  7   Routine = _free               Cycles = 1258    % = 0.917545
  8   Routine = __malloc_at_addr    Cycles = 1182    % = 0.862113
  9   Routine = _malloc             Cycles = 1135    % = 0.827833
 10   Routine = __findbuf           Cycles = 639     % = 0.466066
 11   Routine = __filbuf            Cycles = 639     % = 0.466066
 12   Routine = _fflush             Cycles = 521     % = 0.380001
 13   Routine = _wcp                Cycles = 497     % = 0.362496
 14   Routine = _freopen            Cycles = 443     % = 0.323110
 15   Routine = _printf             Cycles = 400     % = 0.291747
 16   Routine = _strlen             Cycles = 378     % = 0.275701
 17   Routine = __xflsbuf           Cycles = 328     % = 0.239233
 18   Routine = _ipr                Cycles = 321     % = 0.234127
 19   Routine = __findiop           Cycles = 319     % = 0.232668
 20   Routine = _isatty             Cycles = 239     % = 0.174319
 21   Routine = _close              Cycles = 212     % = 0.154626
 22   Routine = __wrtchk            Cycles = 189     % = 0.137851
 23   Routine = _exit               Cycles = 180     % = 0.131286
 24   Routine = _sbrk               Cycles = 137     % = 0.099923
 25   Routine = start               Cycles = 129     % = 0.094089
 26   Routine = _read               Cycles = 106     % = 0.077313
 27   Routine = _ioctl              Cycles = 98      % = 0.071478
 28   Routine = _fopen              Cycles = 86      % = 0.062726
 29   Routine = _finitfp_           Cycles = 75      % = 0.054703
 30   Routine = __exit              Cycles = 63      % = 0.045950
 31   Routine = _fstat              Cycles = 53      % = 0.038657
 32   Routine = _write              Cycles = 53      % = 0.038657
 33   Routine = _open               Cycles = 53      % = 0.038657
 34   Routine = _getpagesize        Cycles = 53      % = 0.038657
 35   Routine = __cleanup           Cycles = 51      % = 0.037198
 36   Routine = cerror              Cycles = 24      % = 0.017505
 37   Routine = fsoft_used          Cycles = 9       % = 0.006564
```

The simulator forks a child task which invokes *wc*. That child task is then single stepped and the simulator disassembles each machine instruction to determine how many cycles are associated with the current instruction. If necessary, the simulator can consult register values as well for conditional branches, etc. The simulator assumes zero wait state memory and the on-chip instruction cache is not used. This provides consistent numbers across all runs.

If the program still contains its symbolic information, then cycle counts per routine are also accumulated, sorted by usage, and printed. Any routine that starts with the characters "CP_" is considered a simulation of hardware and not directly counted in the CPU cycles total. Rather, each "CP_" routine is responsible for incrementing a global variable called *CP_cycles* such that the overall cycle count is incremented by the simulated hardware cycles not by the instructions performing the simulation of that hardware. Hence, there is the *CPU Cycles* value, the *Coprocessor Cycles* value, and the *Total Cycle Count* value, which is the sum of the first two values.

A selection of 8 tasks were implemented in a variety of programs, all of which were run through the simulator on a range of input data. The details of those tasks, the data, and the results are presented next.

5.4. Task Selection

One of the string languages discussed in Chapter 2 is *awk*. At the end of the awk manual there is a set of 8 tasks defined, and the performance of awk on those tasks is compared to a variety of other standard UNIX programs accomplishing the same tasks. In accomplishing these tasks, each of the copying, comparing, and searching string operation categories are thoroughly exercised within the scope of program execution. Because these tasks exercise each category and provide other programs to compare against, they provide a basis for determining the improvements that might be realized by the coprocessor. To accomplish this comparison, we wrote C programs to perform the same tasks and simulated them both with and without the coprocessor. In addition, we vary the amount of input data the tasks must work on to quantify any complexity differences which might appear.

5.5. Task Definition

The 8 tasks are fairly simple. They are enumerated here:

(1) Count the number of lines.

(2) Print all lines containing "root".

(3) Print all lines containing "root", "uucp", or "daemon".

(4) Print the third field of each line.

(5) Print the third and second field of each line, in that order.

(6) Append all lines containing "root", "uucp", and "daemon" to files "jroot", "juucp", and "jdaemon", respectively.

(7) Print each line prefixed by "linenumber: ".

(8) Sum the fourth column of a table.

These task definitions have only been modified by the patterns which are searched for. The data consists of directory and file information as listed by the UNIX "ls -l" command. Each line has the form:

```
drwxr-xr-x  2 bin          1536 Nov 10  1987 bin
```

While the programs listed in the awk manual were run on 10,000 lines (452,960 bytes) of data, we vary the range of data from 100 to 1,000 to 10,000 bytes.

5.6. Results

Each task has a different set of programs to implement it and different string operation requirements. Task 1 counts the number of lines in a file, and eight programs were run to accomplish this task. The *wc* program is a special purpose program provided in the UNIX environment to count lines, words, and bytes in a file. The *fgrep*, *grep*, and *egrep* programs are used for searching and have a parameter to provide a count of the number of lines which contained a string that matched during the search. The *sed* and *awk* string languages are described in Chapter 2. The remaining two programs represent our custom program linked with the system library in one case and linked with our simulated hardware in the other case. The *custom without* program is the linked version *without* the coprocessor. The results for task 1 were:

TABLE 5.2. Task 1 Results

Program	100bytes	1,000bytes	10,000bytes
wc	32,947	156,868	1,392,674
fgrep	23,007	87,842	735,170
grep	29,911	144,862	1,293,516
egrep	43,536	105,183	720,657
sed	28,896	106,883	886,663
awk	132,981	232,963	1,243,501
custom without	20,135	70,554	574,345
custom with	16,125	30,564	174,498

The only string operation that comes into play on this task is *memccpy* which is used by the *gets* function to copy from the input buffer until an end-of-line ('\n') character is encountered. As might be expected for this and all of the tasks, the custom program will always be faster than the more general purpose tools. The simulator results quantify how much faster. Furthermore, the difference in the *custom without* and *custom with* is a direct quantification of the improvement due to the coprocessor. A plot of results for task 1 *custom without* and *custom with* is shown in the following figure.

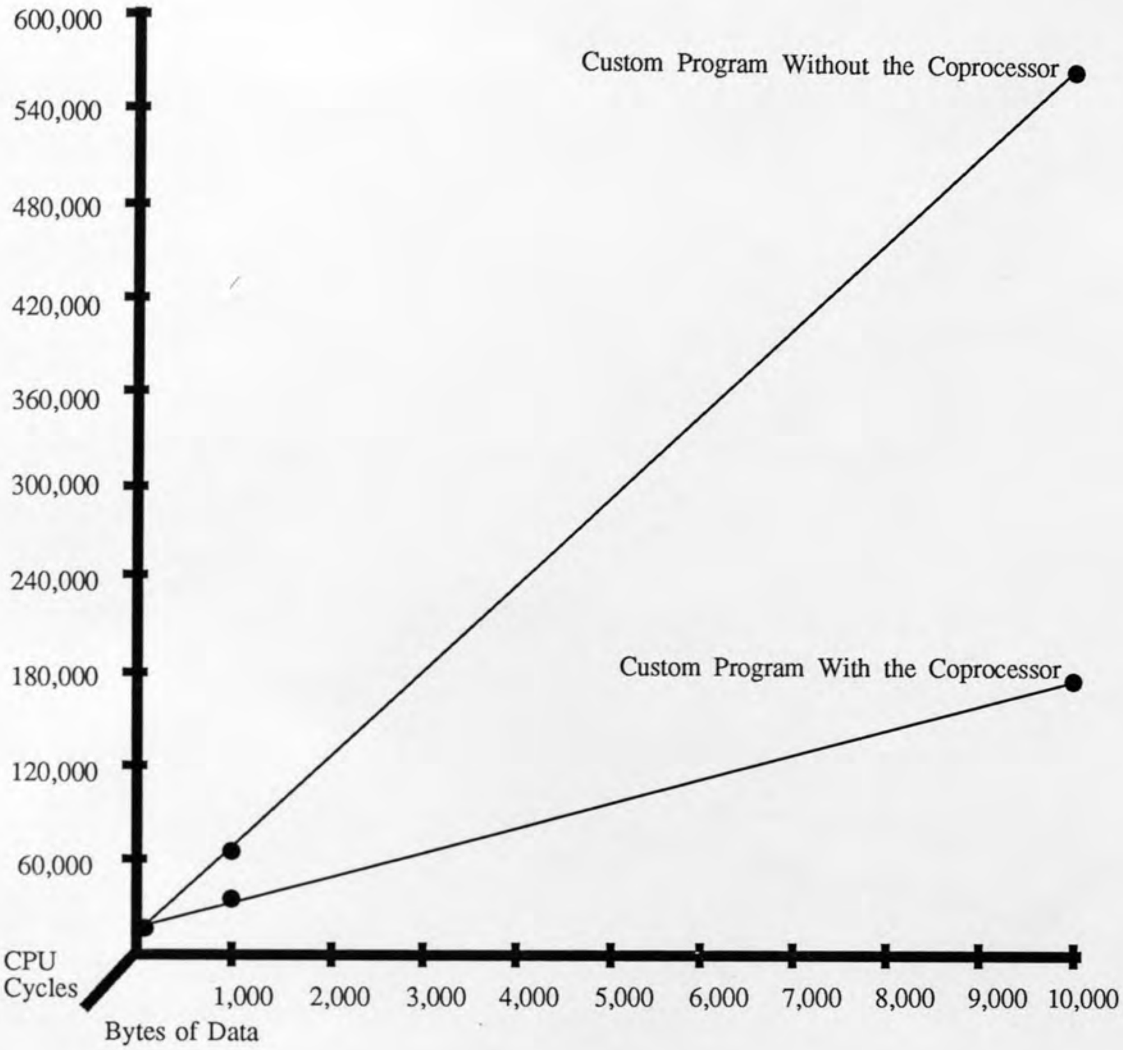


Figure 5.1. Plot of Program Run Times

The nature of the tasks and data yields linear growth in all programs for all 8 tests as the data grows. The tasks which require searching have small exact patterns (length 4 to 6) which are always found in the same position of each line. This effectively renders the algorithm complexity difference negligible and the execution time linear as the amount of data increases. The plots of performance improvement for all 8 tasks would be similar to the one shown for task 1. Only the labels on the *CPU Cycles* axis and the slopes of the lines would change. Because the horizontal axis represents the growth of input data and the vertical axis represents the growth of execution time, the slope of each line is representative of the linear coefficient for execution time. For task 1, the slopes are calculated by:

slope of custom without = (y2-y1)/(x2-x1) = (574,345 - 20,135)/(10,000 - 100) = 56

slope of custom with = (174,498 - 16,125)/(10,000 - 100) = 16

56/16 = 3.5

We can therefore state that task 1 was accelerated by three and a half times through the use of the coprocessor. Appendix C contains the tables for all of the 8 tasks and the details of their results. In each case, the slope of the 100 to 1,000 bytes segment was confirmed to be the same as the slope of the 1,000 to 10,000 bytes segment. Both of those segment/slopes were confirmed to be the same as the 10 to 10,000 bytes slope. In all tasks, linearity and consistent slope were confirmed. The following table summarizes the slopes of the *custom without* and *custom with* program runs for all 8 tasks.

TABLE 5.3. Improvement Per Task

Task	Slope Ratio	String Operations
1	3.5	memccpy
2	2.2	strncmp,memccpy,strlen
3	1.7	strncmp,memccpy,strlen
4	4.9	strpbrk,memccpy,strspn,strtok
5	3.9	strpbrk,memccpy,strspn,strtok
6	1.5	strncmp,memccpy,strlen
7	1.4	memccpy
8	6.0	strpbrk,strspn,memccpy,strtok

As would be expected, the tasks which make use of the searching operations show the best improvement. Both the first and seventh task only make use of memccpy in the coprocessor but have a large difference in improvement. That is due to the percentage of time spent in memccpy for the tasks. Task 1 does little other than read data with memccpy. Task 7 is burdened with printing every line. While tasks 2, 3, and 7 are searching oriented, they make use of the "intuitive" algorithm with strncmp being executed against every character until a match is found or the end of line is encountered. Making the searching algorithm open directly to the programmer as well as the coprocessor would allow these tasks to show considerably better improvement.

5.7. Conclusions

Our simulations have shown that individual operations can achieve a range from double to nine-fold improvement for this CISC architecture. Additionally, the improvements at the programmatic level range from 50% better to 6 times better. Given the nature of the selected tasks, it is not unreasonable to believe that awk or some other string language might well be improved for this architecture by a doubling or tripling in performance.

CHAPTER 6

CONCLUSIONS

We have shown applications which require megabyte to terabyte scale manipulations of non-numeric data. Hardware implementations of special operations are the traditional solution to slow or unacceptable performance. However, recent research in reduced instruction set computers (RISC) tends to indicate that the central processing unit (CPU) is the wrong place to implement the hardware operations. Instead, acknowledging that some complex instructions justify hardware implementations, a coprocessor approach is becoming popular for optionally providing hardware accelerated operations. The intent of this dissertation was to investigate the improvements that might be realized through hardware acceleration of non-numeric operations. We proposed that character string operations were representative of non-numeric operations, and studied the requirements of a set of string languages and their support in software, firmware, and hardware environments.

6.1. Results

The selected operations were identified to fit into the three basic categories of copying, comparing, and searching. Hardware acceleration of copying and comparing yielded a range of improvement from double to nearly nine fold. Searching had even better results through a basic complexity improvement from exponential space and exponential time requirements in a uniprocessor software environment to quadratic space and linear time requirements in our hardware. Since the study of string languages revealed intensive searching requirements, the hardware regular expression searching algorithm presented in Chapter 3 takes on special significance.

The computer industry has been offering optional floating point coprocessors which yield around a three to five fold improvement over software floating point operations. We have demonstrated that comparable results can be obtained for string operations through a fairly simple architecture. Furthermore, we have shown the viability of implementing a non-deterministic algorithm through parallel processor techniques in very large scale integration (VLSI). The value of regular expressions is well known in the computer industry, and our VLSI implementations of the searching algorithms indicate their practicality in the commercial environment.

6.2. Extensions to This Work

The coprocessor was thoroughly simulated and tested for correct functionality. Statistics were then collected on the improvement for a set of programs. However, the simulation had one advantage over an actual implementation. That advantage was the fact that only one process was utilizing the coprocessor and it was effectively non-interruptible. There are only five registers that would need to be saved during an interrupt of a copy or compare operation (i.e., three string registers, plus the SR, IR, and TERM as one register, plus C1 and C2 as another register). Interrupting a search operation would be considerably more difficult due to the amount of information contained in all of the parallel processors (cells).

One solution to this problem of interruptibility might be to have the searching hardware separate from the coprocessor as an allocated resource. In such an environment, the only special consideration required for the hardware searching algorithm would be another control line which indicates to remain idle while waiting for the next data character.

Another consideration not discussed in the simulation is that of concurrency. The CPU might be able to handle some work for another task while the coprocessor is active, thus enabling further improvement in system performance. However, since the coprocessor enters DMA mode and becomes the bus master, interruptibility once again becomes an issue.

Despite the minor inadequacies just discussed about the simulation tool, we believe that the technique we used has some potential for expansion. The concept of forking a child task could be extended to forking several child tasks to simulate a multi-processing environment. Instruction set usage and addressing mode utilization statistics could also be easily obtained.

In the area of searching algorithms, the next natural step is an investigation of the implications of extending the algorithm to recognize context free languages. Additionally, there are a number of related parallel algorithms for non-numeric operations. For example, this university has published algorithms for sorting, longest common sub-sequence (fuzzy searching), and data compression to name a few.

APPENDICES

APPENDIX A

MANUAL ENTRY FOR STRING OPERATIONS

Taken from the AT&T System 5 manual entries for MEMORY(3) and STRING(3) as edited by Sun Microsystems.

NAME

memory, memccpy, memchr, memcmp, memcpy, memset – memory operations

SYNOPSIS

```
#include <memory.h>

char *memccpy (s1, s2, c, n)
char *s1, *s2;
int c, n;

char *memchr (s, c, n)
char *s;
int c, n;

int memcmp (s1, s2, n)
char *s1, *s2;
int n;

char *memcpy (s1, s2, n)
char *s1, *s2;
int n;

char *memset (s, c, n)
char *s;
int c, n;
```

DESCRIPTION

These functions operate as efficiently as possible on memory areas (arrays of characters bounded by a count, not terminated by a null character). They do not check for the overflow of any receiving memory area.

memccpy copies characters from memory area *s2* into *s1*, stopping after the first occurrence of character *c* has been copied, or after *n* characters have been copied, whichever comes first. It returns a pointer to the character after the copy of *c* in *s1*, or a NULL pointer if *c* was not found in the first *n* characters of *s2*.

memchr returns a pointer to the first occurrence of character *c* in the first *n* characters of memory area *s*, or a NULL pointer if *c* does not occur.

memcmp compares its arguments, looking at the first *n* characters only, and returns an integer less than, equal to, or greater than 0, according as *s1* is lexicographically less than, equal to, or greater than *s2*.

memcpy copies *n* characters from memory area *s2* to *s1*. It returns *s1*.

memset sets the first *n* characters in memory area *s* to the value of character *c*. It returns *s*.

NOTE

For user convenience, all these functions are declared in the optional *<memory.h>* header file.

BUGS

memcmp uses native character comparison, which is signed on some machines and unsigned on other machines. Thus the sign of the value returned when one of the characters has its high-order bit set is implementation-dependent.

NAME

string, strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok, index, rindex – string operations

SYNOPSIS

```
#include <string.h>

char *strcat (s1, s2)
char *s1, *s2;

char *strncat (s1, s2, n)
char *s1, *s2;
int n;

int strcmp (s1, s2)
char *s1, *s2;

int strncmp (s1, s2, n)
char *s1, *s2;
int n;

char *strcpy (s1, s2)
char *s1, *s2;

char *strncpy (s1, s2, n)
char *s1, *s2;
int n;

int strlen (s)
char *s;

char *strchr (s, c)
char *s;
int c;

char *strrchr (s, c)
char *s;
int c;

char *strpbrk (s1, s2)
char *s1, *s2;
```

```
int strspn (s1, s2)
char *s1, *s2;

int strcspn (s1, s2)
char *s1, *s2;

char *strtok (s1, s2)
char *s1, *s2;

#include <string.h>

char *index(s, c)
char *s, c;

char *rindex(s, c)
char *s, c;
```

DESCRIPTION

These functions operate on null-terminated strings. They do not check for overflow of any receiving string.

strcat appends a copy of string *s2* to the end of string *s1*. *strncat* appends at most *n* characters. Each returns a pointer to the null-terminated result.

strcmp compares its arguments and returns an integer greater than, equal to, or less than 0, according as *s1* is lexicographically greater than, equal to, or less than *s2*. *strncmp* makes the same comparison but compares at most *n* characters.

strcpy copies string *s2* to *s1*, stopping after the null character has been copied. *strncpy* copies exactly *n* characters, truncating or null-padding *s2*. The result will not be null-terminated if the length of *s2* is *n* or more. Each function returns *s1*.

strlen returns the number of characters in *s*, not including the terminating null character.

strchr (*strrchr*) returns a pointer to the first (last) occurrence of character *c* in string *s*, or a NULL pointer if *c* does not occur in the string. The null character terminating a string is considered to be part of the string.

index (*rindex*) returns a pointer to the first (last) occurrence of character *c* in string *s*, or a NULL pointer if *c* does not occur in the string. These functions are identical to *strchr* (*strchr*) and merely have different names.

strpbrk returns a pointer to the first occurrence in string *s1* of any character from string *s2*, or a NULL pointer if no character from *s2* exists in *s1*.

strspn (*strcspn*) returns the length of the initial segment of string *s1* which consists entirely of characters from (not from) string *s2*.

strtok considers the string *s1* to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string *s2*. The first call (with pointer *s1* specified) returns a pointer to the first character of the first token, and will have written a null character into *s1* immediately following the returned token. The function keeps track of its position in the string between separate calls, so that subsequent calls (which must be made with the first argument a NULL pointer) will work through the string *s1* immediately following that token. In this way subsequent calls will work through the string *s1* until no tokens remain. The separator string *s2* may be different from call to call. When no token remains in *s1*, a NULL pointer is returned.

NOTE

For user convenience, all these functions, except for *index* and *rindex*, are declared in the optional **<string.h>** header file. All these functions, including *index* and *rindex* but excluding *strchr*, *strrchr*, *strpbrk*, *strspn*, *strcspn*, and *strtok*, are declared in the optional **<strings.h>** include file; the reason for this is also historical.

WARNINGS

strcmp and *strncmp* use native character comparison, which is signed on the Sun, but may be unsigned on other machines. Thus the sign of the value returned when one of the characters has its high-order bit set is implementation-dependent.

On the Sun processor, as well as on many other machines, you can *NOT* use a NULL pointer to indicate a null string. A NULL pointer is an error and results in an abort of the program. If you wish to indicate a null string, you must have a pointer that points to an explicit null string. On some implementations of the C language on some machines, a NULL pointer, if dereferenced, would yield a null string; this highly non-portable trick was used in some programs. Programmers using a NULL pointer to represent an empty string should be aware of this portability issue; even on machines where dereferencing a NULL pointer does not cause an abort of the program, it does not necessarily yield a null string.

Character movement is performed differently in different implementations. Thus overlapping moves may yield surprises.

APPENDIX B

REGULAR EXPRESSION PREPROCESSING ALGORITHM

```
/*
 * Curry-Mukherjee Hardware Regular Expression Algorithm Preprocessor
 *
 * compile with:
 *         cc -o reg reg.c
 *
 * run with:
 *         reg "pattern"
 * where "pattern" can have parens "()" alternation "|" closure "*"
 *         fixed length don't care "." and variable length don't care "?"
 *         BE SURE TO PUT QUOTES AROUND THE PATTERN TO PREVENT SHELL
 *         INTERPRETATION OF THE SPECIAL OPERATORS!
 */

#include        <stdio.h>

/*
 *         CELLS            -        max number of hardware cells
 *         LATCHES          -        number of latches per cell
 *         THRESH_HOLD -             start of unprintable characters
 */

#define         CELLS                 256
#define         LATCHES                 5
#define         THRESH_HOLD           127

/*
 * The latches in each cell consist of:
 *         PAT      -        pattern latch
 *         GET      -        get bus value (read latch)
 *         SET      -        set bus value (write latch)
 *         FDC      -        fixed length don't care
 *         VDC      -        variable length don't care
 */

#define         PAT           0
#define         GET           1
#define         SET           2
#define         FDC           3
#define         VDC           4

#define         SUFFIX        0
#define         PREFIX        1
```

```
main(argc,argv)
int     argc;
char    **argv;
{
  /*
   * Variable Usage:
   *     cell_count      -     counter to indicate current cell being modified
   *     closure_flag    -     flag indicating unary closure operator status
   *     pat_ptr         -     counter into the source expression (pattern)
   *     level           -     the depth of parentheses nesting
   *     cell            -     an array containing the hardware latch values
   */

                int     cell_count,closure_flag,pat_ptr,level;
  unsigned      char    cell[LATCHES][CELLS];

  if (argc < 2 || argc > 2)
  {
        fprintf(stderr,"reg: usage: reg 'regular expression'\n");
        exit(1);
  }

  /*
   * initialize the last cell to receive all ending patterns and initialize
   *     all counters and flags.
   */

  cell[PAT][0] = cell[GET][0] = closure_flag = level = 0;
  cell[SET][0] = cell_count = 1;

  /* loop through the entire expression starting with the last character */

  for (pat_ptr=strlen(argv[1])-1; pat_ptr>=0 && cell_count<CELLS; pat_ptr--)
  {
        switch (argv[1][pat_ptr])
        {
          case '*':
                closure_flag = 1;
                break;

          case '(':
                cell[PAT][cell_count] = bus_num(level,closure_flag,PREFIX);
                cell[SET][cell_count] = cell[GET][cell_count] = 1;
                cell_count++;
                bus_num(--level,closure_flag,SUFFIX);
                break;
        case ’)’:
                level++;
                cell[PAT][cell_count] = bus_num(level,closure_flag,SUFFIX);
                cell[SET][cell_count] = cell[GET][cell_count] = 1;
                cell_count++;
                closure_flag = 0;
                break;

        case ’|’:
                cell[PAT][cell_count] = bus_num(level,closure_flag,PREFIX);
                cell[GET][cell_count] = 1;
                if (bus_num(level,closure_flag,SUFFIX) != cell[PAT][cell_count])
                {
                  cell_count++;
                  cell[PAT][cell_count] = bus_num(level,closure_flag,SUFFIX);
                }
                cell[SET][cell_count++] = 1;
                break;

        default:
                if (closure_flag)
                {
                  level++;
                  cell[PAT][cell_count] = bus_num(level,closure_flag,SUFFIX);
                  cell[SET][cell_count] = cell[GET][cell_count] = 1;
                  cell_count++;
                }
                cell[PAT][cell_count] = argv[1][pat_ptr]+THRESH_HOLD;
                switch (argv[1][pat_ptr])
                {
                  case ’?’:      cell[VDC][cell_count] = 1;
                  case ’.’:      cell[FDC][cell_count] = 1;
                }
                cell_count++;
                if (closure_flag)
                {
                  cell[PAT][cell_count] = bus_num(level,closure_flag,PREFIX);
                  cell[SET][cell_count] = cell[GET][cell_count] = 1;
                  bus_num(--level,closure_flag,SUFFIX);
                  closure_flag = 0;
                  cell_count++;
                }
      }
}

if (cell_count >= CELLS)
{
      fprintf(stderr,"reg: regular expression exceeds %d cells.\n",CELLS);
      exit(1);
}
```

```
    cell[PAT][cell_count] = bus_num(level,closure_flag,PREFIX);
    cell[GET][cell_count] = 1;

    printf("input pattern = %s\n",argv[1]);
    reg_printf("pattern = ",cell[PAT],cell_count);
    reg_printf("read    = ",cell[GET],cell_count);
    reg_printf("write   = ",cell[SET],cell_count);
    reg_printf("fldc    = ",cell[FDC],cell_count);
    reg_printf("vldc    = ",cell[VDC],cell_count);

    if (cell[PAT][cell_count] != 1)
    {
        fprintf(stderr,"reg: warning: expression syntax problem.\n");
        exit(1);
    }
}
```

```
/*
 * Procedure:    bus_num
 * Purpose:      management of assigning bus lines
 * Parameters:   level - the current depth of parentheses nesting
 *               closure_flag - indication of scope of closure operator
 *               op - 0 for Suffix request, 1 for Prefix request
 */

bus_num(level,closure_flag,op)
int     level,closure_flag,op;
{
  static          int     bus_stack[CELLS][2],last_level,next_available;

  /* First time through initialize Prefix and next_available */

  if (bus_stack[0][PREFIX] == 0)
  {
        bus_stack[0][PREFIX] = 1;
        next_available = 2;
  }

  /* if nesting goes deeper, new prefix, suffix, and next */
  /* otherwise, if nesting is shallower, pop stack */

  if (level > last_level)
  {
        bus_stack[level][SUFFIX] = next_available;
        if (!closure_flag) next_available++;
        bus_stack[level][PREFIX] = next_available++;
        last_level = level;
  }
  else if (level < last_level) last_level = level;

  return bus_stack[level][op];
}
```

```
/*
 * Procedure:    reg_printf
 * Purpose:      print the cell values in a readable manner
 * Parameters:   label - text to precede the line
 *               array - pointer to which array is to be printed
 *               count - how many elements in the array
 */

reg_printf(label,array,count)
unsigned         char     *label,*array;
                 int      count;
{
  int            i;

  printf(label);
  for (i=count; i>=0; i--)
  {
        if (array[i]<THRESH_HOLD) printf("%3d",array[i]);
        else printf("  %c",array[i]-THRESH_HOLD);
  }
  putchar('\n');
}
```

APPENDIX C

SIMULATION RESULTS

The 8 tasks are:

(1) Count the number of lines.

(2) Print all lines containing "root".

(3) Print all lines containing "root", "uucp", or "daemon".

(4) Print the third field of each line.

(5) Print the third and second field of each line, in that order.

(6) Append all lines containing "root", "uucp", and "daemon" to files "jroot", "juucp", and "jdaemon", respectively.

(7) Print each line prefixed by "linenumber: ".

(8) Sum the fourth column of a table.

Task 1 Results

Program	100bytes	1,000bytes	10,000bytes
wc	32,947	156,868	1,392,674
fgrep	23,007	87,842	735,170
grep	29,911	144,862	1,293,516
egrep	43,536	105,183	720,657
sed	28,896	106,883	886,663
awk	132,981	232,963	1,243,501
custom without	20,135	70,554	574,345
custom with	16,125	30,564	174,498

Task 2 Results

Program	100bytes	1,000bytes	10,000bytes
fgrep	32,280	213,745	1,982,318
grep	34,820	229,681	2,209,356
egrep	111,035	268,888	1,880,414
sed	39,505	222,469	2,132,986
awk	250,379	525,112	3,353,036
custom without	45,545	283,855	2,521,397
custom with	31,696	142,508	1,155,558

Task 3 Results

Program	100bytes	1,000bytes	10,000bytes
egrep	502,521	656,889	2,282,485
sed	73,096	339,196	2,900,337
awk	618,932	926,095	4,033,836
custom without	81,683	530,251	4,380,894
custom with	53,323	316,667	2,589,809

Task 4 Results

Program	100bytes	1,000bytes	10,000bytes
sed	67,082	353,718	3,239,913
awk	174,103	725,003	6,514,343
custom without	34,072	206,738	1,941,985
custom with	17,529	52,869	407,412

Task 5 Results

Program	100bytes	1,000bytes	10,000bytes
sed	71,197	379,057	3,473,716
awk	186,802	757,060	6,739,503
custom without	35,753	221,967	2,091,958
custom with	18,944	65,626	533,477

Task 6 Results

Program	100bytes	1,000bytes	10,000bytes
sed	66,319	319,279	2,882,537
awk	729,549	1,211,451	5,907,195
custom without	88,386	537,218	4,398,600
custom with	64,610	355,338	2,855,815

Task 7 Results

Program	100bytes	1,000bytes	10,000bytes
awk	205,869	854,182	7,395,412
custom without	26,218	140,385	1,305,923
custom with	22,256	100,827	910,372

Task 8 Results

Program	100bytes	1,000bytes	10,000bytes
awk	249,671	829,077	6,905,126
custom without	42,829	268,477	2,532,469
custom with	19,661	56,297	430,566

LIST OF REFERENCES

Aho, Alfred, and Ullman, Jeffrey. *Principles of Compiler Design.* Reading: Addison-Wesley Publishing Co., 1977.

Aho, A., and Corasick, M. "Efficient String Matching: An Aid to Bibliographic Search." *CACM* 18 (June 1975):333-340.

Aho, Alfred V.; Kernighan, Brian W.; and Weinberger, Peter J. "AWK - A Pattern Scanning and Processing Language." Bell Laboratories (September 1978).

Bal, Subash; Chao, George; and Soha, Zvi. "Bilingual, 16-bit microprocessor summons large-scale computer power." *Electronic Design* 2 (January 1980):66-70.

Bal, Subash; Burdick, Ed; Barth, Rick; and Bodine, Dan. "System capabilities get a boost from a high-powered dedicated slave." *Electronic Design* 5 (March 1980):77-82.

Barrett, William A., and Couch, John D. *Compiler Construction: Theory and Practice.* USA: Science Research Associates, Inc., 1979.

Bayer, M.D. "Dialog - An Online Retrieval System for Bibliographic Information." *Proc. COMPCON* (Fall 1978):54-58.

Berenbaum, Alan D.; Condry, Michael W.; and Lu, Priscilla M. "The Operating System and Language Support Features of the BELLMAC-32 Microprocessor." *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems* (March 1982):30-38.

Bird, R. M. "The Associative File Processor, A Special Purpose Hardware System For Text Search and Retrieval." *Proc. IEEE National Aerospace & Electronics Conference* 1 (May 1979):443-449.

Birnbaum, Joel S., and Worley, William S., Jr. "Beyond RISC: High-Precision Architecture." Hewlett-Packard Company (1985):40-47.

Black, D.V. "System Development Corporation's Search Service." *Proc. COMPCON* (Fall 1978):59-64.

Boyer, R., and Moore, J. "A Fast String Searching Algorithm." *CACM* 20 (October 1977):762-772.

Brown, P.J. *Macro Processors and Techniques for Portable Software*. New York: John Wiley & Sons, Inc., 1974.

Coutant, Cary A.; Griswold, Ralph E.; and Wampler, Stephen B. "Reference Manual for the Icon Programming Language, Version 3." *Report TR 80-2*, Department of Computer Science, University of Arizona, 1980.

Crystal, Jeff, and Hendry, Randy. "Non-Numeric Logic Unit and Registers." Class Report, University of Central Florida, October 1983.

Curry, T.; Diaz, E.; Klages, J.; Kotick, D.; and Mukhopadhyay, A. "PAM - A Pattern Matching Chip." *Design Report to MOSIS*, University of Central Florida, 1983.

Curry, Timothy W., and Mukhopadhyay, Amar. "Realization of Efficient Non-Numeric Operations Through VLSI." *VLSI 83*. New York: Elsevier Science Publishers B.V., 1983.

Digital. *VAX11/780 Architecture Handbook*. Maynard: Digital Equipment Corporation, 1977.

Dolotta, T.A.; Haight, R.C.; and Mashey, J.R. "The Programmer's Workbench." *The Bell System Technical Journal* 57 (July/August 1978):2177-2200.

Farber, David J.; Griswold, Ralph E.; and Polonsky, Ivan P. "SNOBOL, A String Manipulation Language." *Journal of the ACM* 11 (January 1964):21-30.

Fischer, M., and Paterson, M. "String-Matching and Other Products." *MIT MAC Technical Memorandum 41* (January 1974):1-21.

Fitzpatrick, D. "VLSI Implementations of a Reduced Instruction Set Computer." *CMU Conference on VLSI Systems and Computations*, Computer Science Press, Carnegie Mellon University (1981).

Floyd, R., and Ullman, J. "The Compilation of Regular Expressions into Integrated Circuits." *Proceedings of 21st annual Symposium on Foundations of Computer Science*, IEEE Computer Society (1980):260-269.

Foster, M., and Kung, H. T. "The Design of Special-Purpose VLSI Chips." *COMPUTER* 13 (January 1980):26-40.

Foster, Mike, and Kung, H.T. "Recognize Regular Languages with Programmable Building-Blocks." *VLSI 81*. San Diego: Academic Press, 1981.

Galil, Zvi. "On Improving the Worst Case Running Time of the Boyer-Moore String Matching Algorithm." *CACM* 22 (September 1979):505-508.

Galil, Zvi. "Optimal Parallel Algorithms for String Matching." Tel-Aviv and Columbia Universities, 1984.

Galil, Zvi, and Seiferas, Joel. "Time-Space-Optimal String Matching." *Journal of Computer and System Sciences* 26 (June 1983):280-294.

Gimpel, J. "A Theory of Discrete Patterns and Their Implementation in SNOBOL4." *CACM* 16 (February 1973):91-100.

Goldberg, R. "Software Design Issues in the Architecture and Implementation of Distributed Text Editors." Ph. D. diss., Rutgers, 1982.

Greenblatt, Richard D. "A Lisp Machine." *Fifth Workshop on Computer Architecture for Nonnumeric Processing, SIGARCH* (March 1980).

Griswold, Ralph E. "Bibliography of Documents Related to the SNOBOL Languages." *Technical Report TR 78-18a*, Department of Computer Science, University of Arizona, September 1979.

Griswold, R.E.; Poage, J.F.; and Polonsky, I.P. *The SNOBOL4 Programming Language*. Englewood Cliffs: Prentice-Hall, 1971.

Harrison, Michael A. *Introduction to Formal Language Theory.* Reading: Addison-Wesley Publishing Co., 1978.

Haskin, R. "Hardware for Searching Very Large Text Databases." Ph. D. diss., University of Illinois at Urbana-Champaign, 1980.

Hennessy, J.; Jouppi, N.; Gill, J.; Baskett, F.; Strong, A.; Gross, T.; Rowen, C.; and Leonard, J. "The MIPS Machine." *Proc. COMPCON* (February 1982):2-7.

Hopcroft, John E., and Ullman, Jeffrey D. *Introduction to Automata Theory, Languages, and Computation.* Reading: Addison-Wesley Publishing Co., 1979.

Horspool, R. Nigel. "Practical Fast Searching in Strings." *Software - Practice and Experience* 10 (1980):501-506.

Hughes, Joan K. *PL/I Structured Programming.* New York: John Wiley & Sons, Inc., 1979.

Karp, Richard, and Rabin, Michael. "Efficient Randomized Pattern-Matching Algorithms." *Report TR-31-81*, Aiken Computation Laboratory, Harvard University, December 1981.

Kernighan, Brian W., and Mcllroy, M. D. *UNIX Programmer's Manual, Seventh Edition.* Englewood Cliffs: Prentice-Hall, 1978.

Kernighan, Brian W., and Ritchie, Dennis M. *The C Programming Language*. Englewood Cliffs: Prentice-Hall, 1978.

Knuth, Donald E. *The Art of Computer Programming - Sorting and Searching*. Reading: Addison-Wesley Publishing Co., 1973.

Knuth, Donald; Morris, J.; and Pratt, V. "Fast Pattern Matching in Strings." *SIAM Journal of Computing* 6 (June 1977):323-350.

Kuck, David J. *The Structure of Computers and Computations.* New York: John Wiley & Sons, Inc., 1978.

Lampson, B.; Paul, M.; and Siegert, H. "Distributed Systems - Architecture and Implementation." *Lecture Notes in Computer Science* (1981).

Larson, S. "Online System for Legal Research." *ONLINE* 1 (July 1977):10.

Larus, J. "A Comparison of Microcode, Assembly Code and High-Level Languages on the VAX-11 and RISC I." *Computer Architecture News* 10 (September 1982):10-15.

Lee, Dik Lun. "ALTEP - A Cellular Processor for High-Speed Pattern Matching." *New Generation Computing* 4 (1986):225-244.

Leerburger, Benedict. "Search and Find." *SKY* (March 1988):52-61.

Linton, Mark A. "Benchmarking Engineering Workstations." *IEEE Design & Test* (June 1986):25-30.

Maller, V. A. J. "The Content Addressable File Store - CAFS." *ICL Technical Journal* (November 1979):265-279.

McCarn, D.B. "Online Services of the National Library of Medicine." *Proc. COMPCON* (Fall 1978):48-53.

Mead, Carver, and Conway, Lynn. *Introduction to VLSI Systems.* Reading: Addison-Wesley Publishing Co., 1980.

Moore, G. "VLSI: Some Fundamental Challenges." *IEEE Spectrum* 16 (April 1979):30-37.

Morgan, T., and Rowe, L. "Analyzing Exotic Instructions for a Retargetable Code Generator." *SIGPLAN Notices* (June 1982).

Motorola. *Introducing the Motorola MC6809*. Austin: Motorola Inc., 1979.

Motorola. *MC68000 16-Bit Microprocessor User's Manua*l. Austin: Motorola Inc., 1980.

Motorola. *MC68010 16-Bit Virtual Memory Microprocessors*. Austin: Motorola Inc., 1983.

Motorola. *MC68000 16/32-BIT Microprocessor Programmers's Reference Manual.* Englewood Cliffs: Prentice-Hall, 1984.

Motorola. *MC68020 32-Bit MicroprocessorUser's Manual.* Englewood Cliffs: Prentice-Hall, 1985.

Moussouris, J.; Crudele, L.; Freitas, D.; Hansen, C.; Hudson, E.; March, R.; Przybylski, S.; Riordan, T.; Rowen, C.; and Van't Hof, D. "A CMOS RISC Processor with Integrated System Functions." *IEEE* (1986):126-137.

Mukherjee, Amar. *Introduction to nMOS and cMOS VLSI Systems Design.* Englewood Cliffs: Prentice-Hall, 1986.

Mukhopadhyay, Amar. "Hardware Algorithms for Nonnumeric Computation." *IEEE Transactions on Computers* C-28 (June 1979):384-394.

Mukhopadhyay, Amar. "Hardware Algorithms for String Processing." *Proc. IEEE International Conference on Circuits and Computers* (October 1980):508.

Mukhopadhyay, Amar. "A Backend Machine Architecture for Information Retrieval." In *Research and Development on Information Retrieval*, 296-309. London: Butterworth & Co. Ltd, 1981.

National Semiconductor. *NS16000 Programmer's Reference Manual.* Santa Clara: National Semiconductor Corporation, 1981.

O'Dowd, Dan; Kohn, Les; and Soha, Zvi. "Architecture of 16-bit micro underpins VLSI computing." *Electronic Design* 28 (June 1980):171-176.

Ossanna, Joseph F. *Nroff/Troff User's Manual.* Murray Hill: Bell Laboratories, 1976.

Radin, G. "The 801 Minicomputer." *Proc Symposium on Architectural Support for Programming Languages and Operating Systems* (March 1982):39-47.

Rector, Russel, and Alexy, George. *The 8086 Book.* New York: OSBORNE/McGraw-Hill, 1980.

Roberts, D. "A Computer System for Text Retrieval: Design Concept Development." *Report RD-77-10011*, Central Intelligence Agency, 1977.

Salomaa, Arto. *Theory of Automata.* Elmsford: Pergamon Press, 1969.

Sites, Richard L. "How to Use 1000 Registers." *Proceedings of the Caltech Conference on VLSI*, (January 1979):527-532.

Sprowl, J.A. "Computer Assisted Legal Research." *American Bar Foundation Research Journal* (1976):175-227.

Sun Microsystems. *A RISC Tutorial.* Mountain View: Sun Microsystems, Inc., 1987.

Thompson, Kenneth. "Regular Expression Search Algorithm." *CACM* 11 (June 1968):419-422.

Trickey, H. W. "Good Layouts for Pattern Recognizers." *IEEE Transactions on Computers* 31 (June 1982):514-520.

Vickers, Frank D. *Introduction to Machine and Assembly Language: Systems/360/370.* New York: Holt, Rinehart and Winston, 1971.

Wetherell, Charles. *Etudes for Programmers.* Englewood Cliffs: Prentice-Hall, 1978.

Winston, Patrick Henry. *Artificial Intelligence*. Reading: Addison-Wesley Publishing Co., 1979.

Winston, Patrick Henry, and Horn, Berthold Klaus Paul. *LISP*. Reading: Addison-Wesley Publishing Co., 1981.

Yianilos, Peter. "A Dedicated Comparator Matches Symbol Strings Fast and Intelligently." *Electronics* (December 1983):113-117.

Zilog. *Z80-Assembly Language Programming Manual*. Campbell: Zilog, Inc., 1978.

Zilog. *Z8001 CPU, Z8002 CPU Product Specification*. Campbell: Zilog, Inc., 1979.