OASiS :
An Open Architecture Sieve System for Problems in Number Theory

by

Allan J. Stephens

A thesis
presented to the University of Manitoba
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in
the Department of Computer Science

Winnipeg, Manitoba

ISBN 0-315-63218-6

OASiS:

AN OPEN ARCHITECTURE SIEVE SYSTEM FOR PROBLEMS IN NUMBER THEORY

BY

ALLAN J. STEPHENS

A thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements of the degree of

DOCTOR OF PHILOSOPHY

(This page contains form approving the thesis.)

I hereby declare that I am the sole author of this thesis.

I authorize the University of Manitoba to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Manitoba to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Allan J. Stephens

Abstract

The technique of *sieving* has been known since the time of Eratosthenes, and has been used to solve problems involving systems of linear congruences since the end of the eighteenth century. As a wide range of number theoretic problems can be converted into sieving problems, considerable effort has been expended in constructing machines which are fast enough to solve problems that are otherwise intractable. Most notable is the work of D. H. Lehmer, who pioneered the development of automatic sieving hardware in the 1920's. His sieving systems have made use of such diverse technologies as bicycle chains, photoelectric gears, and general-purpose computers. Lately, attention has shifted to machines using integrated circuits such as high-speed shift registers.

The latest development in automated sieving is the "Open Architecture Sieve System" (OASiS). The system features a specially-designed computer—the Open Architecture Sieve—that is capable of testing possible solutions to a system of linear congruences at a rate of over 200 million numbers per second. Unlike current sieves, the OAS can assume a variety of configurations, allowing the user to alter the number of congruences being tested in hardware and their size. The sieve can be easily upgraded to accommodate more and larger congruences; additional sieve processors can also be installed to permit sieving to be performed in parallel. The OAS runs as a peripheral to a conventional minicomputer which runs special software that oversees the execution of sieving problems. This software increases the speed at which many sieving problems are solved by automatically optimizing sieving whenever one or more congruences with a single acceptable residue class are present.

This thesis discusses the generalized sieving problem and the history of automated sieving systems. It then introduces the basic concepts behind the design of the Open Architecture Sieve System. This is followed by chapters describing the operation of the system software and the OAS hardware, and subsequent chapters detailing their design and construction. The thesis then presents some results of problems that have been run using OASiS. This includes extending the tables of known pseudo-squares and pseudo-cubes, finding periodic continued fractions with long periods, and finding polynomials which have a high density of prime values.

Acknowledgements

The author was assisted by many individuals during the development of this thesis. Although there is room to mention only a few by name, the efforts of all those who were involved are greatly appreciated.

Foremost among those who worked on OASiS itself was our hardware technician, Dale Brodland. Dale designed the parts of the Open Architecture Sieve that I did not have the experience to do myself, and was instrumental in debugging the hardware once it was built. Without his help, the OAS might never have become fast enough or reliable enough to be a successful research tool. Thanks also go out to Dr. F. J. Burkowski of the University of Waterloo, who helped to establish the foundations of the OAS design.

The primary financing for OASiS came from my advisor, Dr. Hugh C. Williams, who faithfully supported the system throughout the long interval from initial concept to finished product. A University of Manitoba Grant in Aid of Research provided funding for a prototype version of the sieve, and its support is also gratefully acknowledged. The Natural Sciences and Engineering Research Council of Canada contributed to OASiS in the form of a postgraduate scholarship to the author.

Special thanks are directed to Dr. Derrick Henry Lehmer who provided many of the interesting and informative details about the early sieving machines described in Chapter 1. Dr. Lehmer kindly invited me into his home and patiently answered an almost endless series of questions about his work, allowing me relate information that has never been formally recorded anywhere. I am also grateful to Dr. John Brillhart, his former pupil, who pointed out several errors in this chapter and suggested additional references. My thanks are also extended to the members of my thesis examination committee, who provided a similar valuable service in their critique of the entire document.

Finally, I must thank the people whose contributions to OASiS, although indirect, were nonetheless essential to its success. Foremost among these is my loving wife, Billie, who stuck by me throughout the years of work on my doctorate, and frequently helped shoulder the burden when things got tough. Her patience, understanding, and support were greater than I had any right to expect. My thanks also go out to those many friends who provided a sympathetic ear or an encouraging word when I needed it most.

Table of Contents

Chapter 1 :
An Introduction to Sieving

Attempt the end, and never stand to doubt;
Nothing's so hard, but search will find it out.

Robert Herrick, *Seek and Find*

The technique of sieving has been used to solve mathematical problems since the time of Eratosthenes (fl. 230 B.C.). Although it is a general method that can be applied to a variety of problems in number theory, sieving is also computationally intensive; as a result, its applications are bounded by the rate at which the required operations can be performed. Since the late eighteenth century, a variety of devices have been constructed to increase the speed of sieving, ranging from simple mechanical aids that improve the efficiency of sieving by hand to high speed computer systems that do sieving automatically. This chapter describes the generalized sieving problem and some of its applications, and traces the development of mechanized sieving over the last 200 years.

1.1 THE SIEVING PROBLEM

Any mathematical problem that requires finding solutions to an arbitrary system of linear congruences involves an instance of the *generalized sieving problem* (GSP). In general, a sieving problem P defines k linear congruences,

$$x \equiv r_{i1}, r_{i2}, \ldots, r_{in_i} \pmod{m_i} \quad \begin{cases} i = 1, 2, \ldots, k \\ 1 \le n_i \le m_i \end{cases},$$

where the moduli $m_1, m_2, \ldots, m_k$ are positive integers. It may be assumed that the m_i are relatively prime in pairs and that each set of admissible residues

$$R_i = \{ r_{i1}, r_{i2}, \ldots, r_{in_i} \}$$

contains distinct, non-negative integers less than m_i. The solution set $S(P)$ for P is defined to be all $x \in \mathbf{Z}$ that lie within an interval specified by P, say $A \le x < B$, satisfy all k congruences, and satisfy any additional restrictions placed on x by P. Depending on the precise requirements of the problem, solving P may entail generating $S(P)$, finding a subset of $S(P)$, or simply determining the number of elements in $S(P)$.

As an illustration of a sieving problem, consider a P that utilizes the following set of five congruences:

$$\begin{array}{ll} x \equiv 1 & \pmod 8 \\ x \equiv 1 & \pmod 3 \\ x \equiv 1,4 & \pmod 5 \\ x \equiv 1,2,4 & \pmod 7 \\ x \equiv 1,3,4,5,9 & \pmod{11}, \end{array}$$

the restriction $x \neq n^2$, and the range $0 \leq x < 3000$. This problem arises naturally from the study of quadratic residues (see Chapter 7). In this instance, there are two solutions in $S(P)$: 2641 and 2689.

1.1.1 Solving a Sieving Problem

The entries of $S(P)$ are the result of an exceedingly complex function of m_i, r_{ij}, A, B, and the additional restrictions. The simplest algorithm for finding them is to treat each congruence as a "screen" or "sieve" which lets through only those integers which lie in the acceptable residue classes. Finding all of the x values that solve the k congruences then involves taking each integer in $[A,B)$, applying it to each of the k sieves in turn, and putting it in $S(P)$ if it makes it through all of the sieves. Any additional restrictions specified by P can be imposed by using a further sieve that rejects any value lacking the required properties. This method of solving the problem is what gives it its name.

The sieving method can be implemented in a variety of ways, ranging from a strictly sequential approach to a massively parallel one. Opportunities for parallel processing arise because the final acceptance or rejection of a given x does not depend on the order in which the sieves are applied or on the results of sieving any other value in the search interval. It is theoretically possible to test a given integer against all of the sieves simultaneously, to test all of the integers in $[A,B)$ against a given sieve simultaneously, or both. In practice, the amount of parallelism that can be employed is limited by implementation factors, which means that some form of serial processing must be performed when these limits are exceeded. A typical sieving implementation tests elements of the specified interval in sets of t, starting with $A, A+1, \ldots, A+t-1$. Each set of t values is applied simultaneously to as many sieves as possible, and then to the remaining sieves in a sequential manner. If a point is reached where all t entries have been rejected, the remaining sieves need not be applied. This approach limits the number of values tested in parallel to a feasible number, and also allows the system to handle "search" problems, in which the upper bound of the interval $[A,B)$ is unknown. Such problems arise frequently in number theory, and usually involve finding the n smallest elements of $S(P)$. Sieving terminates either when the search interval has been completely processed or when n solutions have been found.

1.1.2 The Complexity of Sieving

The simple, deterministic nature of the sieving process makes it an attractive method for solving the GSP. Unfortunately, a comparison of the running time of the algorithm against the size of the problem shows that sieving is not a polynomial time algorithm. The specification of an arbitrary sieving problem that has no additional restrictions on its solutions defines k congruences and a search interval $[A,B)$, and can be represented using

$$\sum_{i=1}^{k} m_i + \log_2 A + \log_2 B$$

bits. If testing a given x value to see if it satisfies a single congruence is considered to be a basic operation, then finding all entries of $S(P)$ may require up to

$$k(B-A)$$

operations to test all of the values in $[A,B)$ against all of the problem's congruences. This is an exponential function of the problem's size. The same is true for search problems, which use an open-ended interval ($B = \infty$). Here, determining $S(P)$ can require up to

$$k \prod_{i=1}^{k} m_i$$

operations. After this point the pattern of solutions repeats with a period equal to the product of the moduli of the congruences.

The exponential nature of the sieving algorithm means that it is easy to construct problem instances that are too large to be solved within a reasonable amount of time, so it would be nice to find a better way to solve an arbitrary system of congruences. Unfortunately, not only have mathematicians been unable to find a polynomial-time, deterministic algorithm that does this, but not even a polynomial-time *non*-deterministic algorithm is known! For example, if the claim is made that a particular sieving problem has

$$S(P) = \{ x_1, x_2, \ldots, x_s \} ,$$

it is easy to verify that the x_i are solutions. However, the only method known for showing that there are no others is to test the remaining values in $[A,B)$ to show that they are not solutions. Thus, the task of verifying a proposed $S(P)$ is just as difficult as generating it by sieving. Similarly, a claim that y is the smallest solution in $S(P)$ can only be verified by testing all of the values less than y.

The failure to find a computationally efficient method for solving the GSP is a result of its intrinsic difficulty. Patterson [Pat89] has recently shown that the NP-complete problem of "quadratic congruences" [MA78] can be transformed into a sieving problem,

demonstrating that the GSP is NP-hard—that is, at least as hard as the problems in NP. Thus, it will probably be some time before the "brute force" technique of sieving will be replaced by a better algorithm for solving arbitrary sieving problems. Since there is currently no good method for verifying the solutions to a sieving problem, there is no proof that the GSP even lies in NP at all. It may lie in a class of problems that are even more difficult than NP, making the task of finding an improved algorithm even harder.

Even though the sieving method is an exponential algorithm, it does not mean that sieving cannot be used to solve non-trivial sieving problems. From a human standpoint, a sieving problem with the interval $[0,2B)$ can be considered twice as difficult as the same problem covering $[0,B)$ because there are twice as many possible solutions; the same is true if $2k$ congruences are involved instead of k congruences, since there are twice the number of restrictions on its solutions. If this "human" notion of problem size is used, rather than the normal "bit counting" approach, then the time required to solve a problem by sieving is directly proportional to its size. This, along with the fact that the simple, deterministic nature of sieving allows it to be implemented efficiently, means that sieving can be utilized to solve many significant problems in number theory.

1.1.3 Applications of Sieving

The sorts of sieving problems that arise in number theory are quite varied in nature, and several date back many centuries. One is related to the well-known *Sieve of Eratosthenes* (see [Dic19]) and uses the first k primes $(p_1, p_2, \ldots, p_k)$ as moduli and whose residue sets, R_i, exclude only zero. Such a system of congruences removes only multiples of the primes used, so sieving the interval $[p_k+1,p_{k+1}^2)$ generates all primes lying in that range. For example, the system of congruences obtained using $k = 4$ is

$$\begin{array}{ll} x \equiv 1 & (\text{mod } 2) \\ x \equiv 1,2 & (\text{mod } 3) \\ x \equiv 1,2,3,4 & (\text{mod } 5) \\ x \equiv 1,2,3,4,5,6 & (\text{mod } 7)\,, \end{array}$$

and produces the primes from 11 to 113 when the interval [8,121) is sieved.

At the opposite end of the spectrum is the *Chinese Remainder Problem*, in which only one residue class is acceptable for each congruence. Such problems have exactly one solution modulo $m_1m_2\ldots m_k$. For example, the system

$$\begin{array}{ll} x \equiv 1 & (\text{mod } 2) \\ x \equiv 2 & (\text{mod } 3) \\ x \equiv 3 & (\text{mod } 5) \end{array}$$

has the single solution $x \equiv 23 \pmod{30}$. The CRP is notable because it is one of the few classes of sieving problem which can be solved by a means other than sieving. One simple approach repeatedly merges pairs of congruences by means of the Euclidean Algorithm until only one congruence remains (see §4.3.2 of [Knu81]).

A much larger class of sieving problems, and one which is of considerable interest to mathematicians, involves finding integers x and y which solve $f(x,y) = 0$, where f is a polynomial of arbitrary degree with integer coefficients of arbitrary size. The "method of exclusion" of Gauss allows us to convert this Diophantine equation to a set of necessary (but not sufficient) sieve conditions which can be solved for x by sieving. The solutions to the original equation are then found among the few x values that remain instead of the entire solution space of possible x and y values. The lack of restrictions on f allows this technique to be applied to a wide range of problems in number theory. A brief list of such problems has been given by Lehmer [Leh66] and includes:

- finding the representations of a large number by a given binary quadratic form, for example $N = x^2 - y^2$,
- finding the binomial units of a given algebraic number field,
- finding the numbers (or the number of numbers) for a given polynomial that lie between given limits.

As an illustration of this technique, a method of factoring an arbitrary large integer is given below. Although based on Fermat's idea of writing the integer as the difference of two squares [Dic19], the actual algorithm was developed by Gauss ([GC66], articles 319-325). A more complete discussion of both approaches is given by Brillhart [Bri81].

Suppose that we wish to factor an odd integer, $n > 0$. If n is composite, then it must be the product of two odd factors, say $n = U \cdot V$, where $U \geq V$. If we define

$$x = \tfrac{1}{2}(U+V)\ ,\ y = \tfrac{1}{2}(U-V)$$

and solve for U and V, we find

$$U = (x + y)\ ,V = (x - y)$$

and thus,

$$n = (x + y)(x - y) = x^2 - y^2$$

or

$$y^2 = x^2 - n\ .$$

Therefore, if we can find an x value for which $x^2 - n$ is a perfect square, we can calculate the factors of n, U and V. The number of such x values that exist is equal to the number of ways that n can be represented as a product of two non-trivial factors.

Suppose we have an x such that $y^2 = x^2 - n$. Then $y^2 \equiv x^2 - n \pmod{p}$ for *any* $p \in \mathbf{Z}$, meaning $x^2 - n$ must be a quadratic residue modulo p. If p is an odd prime which does not divide n, only $\frac{1}{2}(p + (n/p))$ of the p residue classes to which x may belong result in $x^2 - n$ being a quadratic residue.[1] We therefore construct

$$x \equiv r_1, r_2, \ldots, r_k \pmod{p}, \qquad k = \tfrac{1}{2}(p + (n/p))$$

which specifies these residue classes and start sieving. The sieving does not guarantee that each x found results in $x^2 - n$ being a perfect square, but it does exclude many x values for which the expression is not a square.

The chances of finding a suitable x is improved greatly by applying this approach to a number of different p values. This produces a set of linear congruences, each of which excludes roughly half of its residue classes. If k congruences are used, only about one in 2^k numbers will make it through the sieving process to require the test to see if $x^2 - n$ is a perfect square. Since finding an x whose corresponding $x^2 - n$ is a quadratic residue of all k primes but is not a perfect square is relatively unlikely, most x values that solve the congruences allow us to factor n once k becomes large. The effectiveness of the sieving can be improved significantly by forming congruences modulo p^2 rather than modulo p since this results in a lower proportion of acceptable residue classes.

The interval to be searched can be bounded easily . We can arbitrarily restrict $x > 0$, since if $x^2 - y^2 = n$, then $(-x)^2 - y^2 = n$ as well. Thus, $x \geq \lceil \sqrt{n} \rceil$. As n is odd, we know that the second term in the factorization $(x + y)(x - y) = n$ must be at least 3, meaning the first term (and therefore x) must be $\leq \lfloor \frac{n}{3} \rfloor$. If this range is exhausted without finding a factorization of n, then n is prime.

As an example of this procedure, let us factor $n = 5917$ using the primes 3, 5, and 7. We first construct a table to determine the residue classes to be examined.

p	$x \pmod{p}$	$x^2 \pmod{p}$	$n \pmod{p}$	$x^2 - n \pmod{p}$
3	0,1,2	0,1,1	1	2,0,0
5	0,1,2,3,4	0,1,4,4,1	2	3,4,2,2,4
7	0,1,2,3,4,5,6	0,1,4,2,2,4,1	2	5,6,2,0,0,2,6

The underlined values in the final column of the table signify the residue classes which cannot contain perfect squares. The set of congruences to be used is therefore

[1] (a/b) denotes the Legendre symbol.

$$\begin{aligned} x &\equiv 1,2 \pmod 3 \\ x &\equiv 1,4 \pmod 5 \\ x &\equiv 2,3,4,5 \pmod 7 . \end{aligned}$$

The interval to be searched goes from $A = \lceil \sqrt{5917} \rceil = 77$ to $B = \lfloor \frac{5917}{3} \rfloor + 1 = 1973$. There are 289 solutions for x in this range, only one of which leads to a factorization. At $x = 79$ we find that $79^2 - 5917 = 324 = 18^2$, giving us the result

$$U = 79 + 18 = 97 \, , \, V = 79 - 18 = 61 \quad \Rightarrow \quad 5917 = 61 \cdot 97.$$

All of the other solutions for x produce expressions which are not perfect squares; for example, the solution at $x = 86$ results in $86^2 - 5917 = 1479 \approx (38.45777)^2$. The unusually high proportion of "non-factoring" solutions to "factoring" solutions in this example is primlarily due to the very small number of congruences used during sieving, and could be easily decreased by using more moduli.

The factoring algorithm is typical of applications of sieving—its usefulness is entirely dependent on the ability of the user to find solutions to a set of congruences quickly. When an efficient method exists, 15 digit numbers can be factored in less than a day even though the algorithm's running time is $O(n)$. To factor values of up to 25 digits, a modified version of the factoring algorithm can be used whose running time is $O(\sqrt{n})$ [LL74]. Although much better factoring algorithms now exist that under suitable heuristics are of complexity $O(e^{\sqrt{\log n \log\log n} + o(1)})$ [Pom89], until 1970 the most efficient means known of factoring an arbitrary large integer was to use a sieving approach [MB75].

1.2 THE DEVELOPMENT OF SIEVE AUTOMATION

Mathematicians have used the technique of sieving as a tool for solving problems in number theory for over 200 years. During that time, a variety of mechanisms have been developed to increase the speed and accuracy of the sieving process. Unfortunately, those who utilized sieving often published the results of their work with little or no description of the methods they used to obtain them. Some may have felt that little could (or should) be said about a technique that amounted to little more than a brute force search; others may have found the effort of publishing their methods too great to bother with given the limited interest in sieving among mathematicians and computing specialists. Whatever the reason, the result is that much is unknown about many of the sieving systems used, and what *is* known has been pieced together from a variety of sources, some of which give conflicting accounts. This section amalgamates this material into a single account that highlights the most significant developments in automated sieving.

1.2.1 Sieving By Hand

The earliest attempt to apply a mechanical technique to solving the GSP was made by Legendre in 1794 [Rub83]. While his *strip method* increased the speed and accuracy of sieving, it was still a rather slow and inconvenient manual technique. Even so, it remained the best method for solving a GSP until well into the 20th century.

It is possible that the inspiration for Legendre's invention came from an earlier application of mechanical devices to sieving: the construction of *factor tables*. Such tables list the prime factors, or in some cases just the smallest prime factor, for every integer in a specified range, and were widely used by mathematicians in the pre-computer age. Not only did such a table provide a rapid test for properties directly related to a number's factorization, such as primality or the presence of square factors, but it was often helpful in ordinary mathematical calculations, say, allowing the user to obtain a more accurate logarithm for a number by summing the logarithms of its factors. Numerous factor tables were constructed between the 17th and 20th centuries, starting in 1659 with a table by Rahn (or Rhonius) extending up to 24,000; a list of early factor tables and their formation is described by J. W. L. Glaisher [Gla78].

The creation of a factor table is a relatively simple process that is based on the sieve of Eratosthenes [Leh18]. All of the integers in the desired interval are written down in a line, then a "2" is written underneath every even number, a "3" under every value divisible by 3, and so on; the table is complete when all primes less than the square root of the upper bound of the table have been processed in this manner. The table can be reduced in size by about 75% by arranging the numbers in rows of 30 and deleting the 22 columns containing the values divisible by 2, 3, or 5; no significant information is lost since any number divisible by these factors can be spotted easily without using the table.

Although a number of early factor tables were created entirely by hand, the fact that multiples of a prime p are separated by exactly p units means that the process of sieving out multiples is very regular. The first to take advantage of this was C. F. Hindenburg, who in the mid-1770's utilized a strip of thick paper with a regular pattern of holes (*patrone*) to aid in identifying the multiples of p. A machine containing a series of rods was also used, and is described by Bernoulli [Ber85]. Hindenburg's table was never published. At roughly the same time, Anton Felkel independently utilized the same approach in beginning the construction of a table that was to extend up to ten million. The Austrian government financed the publication of the portion up to 408,000 in 1776, but when it sold poorly

(probably due to its unorthodox layout) the entire edition was scrapped to make cartridges for use in the war against the Turks. Only a few copies were saved. Later mathematicians used mechanical aids to produce tables that were more successful. In 1817, Burckhardt used paper stencils to generate and published a table of factors for the integers up to three million; by 1883, similar tables by Glaisher[2] and Dase had continued the work up to nine million [Leh18]. Finally, in 1909, D. N. Lehmer produced a factor table (and a smaller table of primes) that extended up to ten million, reaching Felkel's goal over 130 years after it was first proposed.

Beginning in 1924, Lehmer used a different form of sieving to factor large integers: the *factor stencil* [LE39]. The technique is based on the principle that if a given integer R is a quadratic residue of N, then it is also a quadratic residue of all of the factors of N. Each stencil represented a different R, and had 5000 cells corresponding to the first 5000 primes. A hole was punched in a given cell only if R is a quadratic residue for the associated prime. Factoring was accomplished by finding a small set of quadratic residues for N and superimposing the stencils for those R; only if all cells in a given position contained a hole could the corresponding prime possibly divide N. Trial division could then be used to show whether any of these primes were factors. For a 10 digit number, less than a dozen quadratic residues were usually sufficient to cut the number of possible factors down to 2 or 3. Although more complicated than referring to a factor table, the factor stencil method had an important advantage: it still gave useful information even if the number to be factored was larger than the stencils were designed to handle. Lehmer's first stencils were constructed largely by hand and only a few copies of the entire set were made, but in 1939 Elder constructed a revised and extended version of the stencils using Hollerith cards; these were easily reproduced and numerous sets were distributed by the Carnegie Institution.

Despite the increases in speed and accuracy that these approaches gave, sieving remained a cumbersome technique. One problem was that the work was still done by hand and remained a relatively slow and error-prone process. As well, it was necessary to decide on the limits of the interval to be processed before sieving began. This was no drawback for problems such as the creation of factor tables since the entire interval was always processed regardless of the results obtained, but it presented a major handicap when applied to the GSP. If, for example, the smallest value satisfying a system of congruences lay near the start of the search interval, the entire interval was sieved using all but the last

---

2 The father of J. W. L. Glaisher.

congruence before the solution was discovered; all of the effort involved in sieving the high end of the interval was wasted. It was not until the advent of the first automatic sieving machine that these difficulties were eliminated and sieving became practical on a large scale.

1.2.2 Principles of Sieve Automation

The periodic behaviour of the sieving process naturally lends itself to the construction of a machine that can solve the GSP. Consider the task of finding the non-negative solutions to an arbitrary set of k linear congruences

$$x \equiv r_{ij} \pmod{m_i} \qquad \begin{cases} i = 1, 2, \ldots, k \\ j = 1, 2, \ldots, n_i < m_i \end{cases}.$$

The solutions can be found by constructing a table of $m_1 m_2 \ldots m_k$ columns numbered from 0 to $m_1 m_2 \ldots m_k - 1$. For each congruence, add a row to the table which indicates which values lie in the set of acceptable residue classes, R_i. Any column of the table which satisfies all of the congruences represents a solution to the congruences.

			x - 0 1 2 3 4 5 6 7 8 ...
	$x \equiv 0$	(mod 2)	mod 2 - $\surd_0$ $\times_1$ $\surd_0$ $\times_1$ $\surd_0$ $\times_1$ $\surd_0$ $\times_1$ $\surd_0$ ...
(a)	$x \equiv 1,2$	(mod 3)	mod 3 - $\times_0$ $\surd_1$ $\surd_2$ $\times_0$ $\surd_1$ $\surd_2$ $\times_0$ $\surd_1$ $\surd_2$ ...
	$x \equiv 0,2,3$	(mod 5)	mod 5 - $\surd_0$ $\times_1$ $\surd_2$ $\surd_3$ $\times_4$ $\surd_0$ $\times_1$ $\surd_2$ $\surd_3$ ...
			x - 0
	$x \equiv 0$	(mod 2)	mod 2 - $\surd_0$ $\times_1$
(b)	$x \equiv 1,2$	(mod 3)	mod 3 - $\times_0$ $\surd_1$ $\surd_2$
	$x \equiv 0,2,3$	(mod 5)	mod 5 - $\surd_0$ $\times_1$ $\surd_2$ $\surd_3$ $\times_4$
			x - 3
	$x \equiv 0$	(mod 2)	mod 2 - $\times_1$ $\surd_0$
(c)	$x \equiv 1,2$	(mod 3)	mod 3 - $\times_0$ $\surd_1$ $\surd_2$
	$x \equiv 0,2,3$	(mod 5)	mod 5 - $\surd_3$ $\times_4$ $\surd_0$ $\times_1$ $\surd_2$

Figure 1-1. Principles of Sieve Automation

Figure 1-1 (a) shows a portion of a table for an example involving three congruences; the solutions at 2 and 8 are readily apparent. In practice, this sort of table is too large to construct, but since each row of the table exhibits a cyclic pattern corresponding to the list

of acceptable and forbidden residue classes for its associated congruence, the information in the table can be compressed to a small fraction of its original size by recording the pattern of residues only once (Figure 1-1 (b)). Any column of the original table can be generated by shifting each of the rows to the left to advance the appropriate set of residues to the first column; the leftmost residue wraps around to the rightmost end. For example, three shifts generates column 3 of the original table (Figure 1-1 (c)). In general, column t can be obtained by performing t shift operations.

From here, it is easy to see how a machine that performs sieving can be constructed. Each congruence used in the problem is represented by a loop, or *ring*, whose size corresponds to its modulus. The ring for congruence i is built with m_i positions, and tags are placed in those slots representing acceptable residue classes. The machine examines one position from each ring at a fixed location called the *tap*, or *window*, and simply advances the rings in unison until all of the rings present an acceptable residue simultaneously. A *trial counter* records the number of shifts performed by the machine, and is used to determine the value of each solution found. The machine can be set up to start searching from any point A by setting ring i so that position $A \pmod{m_i}$ is in the window when the machine is turned on; after s shifts, the value being tested is $A+s$. Any machine of this type is known as a *sieve*.

An *automatic sieving system* consists of a sieve, a means of loading problems into the sieve, and a means of recording the solutions found by the sieve. The recording stage does not necessarily have to note the exact value of each solution, since some problems only require that the number of solutions be determined; in such cases, the recording stage can simply increment a counter each time a solution appears. The loading and recording parts of the system may be implemented by machines or by a human operator; the only portion of the system that must be entirely automatic is the sieving process itself. The construction of a completely automatic sieving system that can run problems without human intervention is a relatively recent development in the history of automated sieving.

The sieve design provides the user with a fast and accurate means of solving the GSP. Since the search is entirely automatic, it can be performed at high speed for long periods without error. By altering the placement of tags of the rings, the sieve can be loaded with any system of congruences which corresponds to its collection of rings. The finite size of the device does means that some sets of congruences cannot be implemented by the sieve completely, but this apparent drawback can be overcome with little difficulty in most common problems. (See §1.2.2.2, "Filtering Solutions", below.) The application of such

a machine to sieving also overcomes the flaws of hand sieving mentioned previously: it is efficient and, since it simultaneously tests each solution candidate against all of the congruences rather than each congruence against all of the solution candidates, solutions that occur near the beginning of the search interval are discovered quickly.

1.2.2.1 OPTIMIZATIONS

The performance of the basic sieve model can be increased significantly by testing multiple solution candidates in parallel. There are two ways of increasing the sieving rate by a factor of t: by using t sieves or by constructing a single sieve with a t position solution window that tests t consecutive positions from each ring and then advances the rings by t positions. This multi-tap approach is relatively cheap to implement since it is basically a single tap sieve with additional solution detection hardware; in contrast, a set of single tap machines involves the duplication of the complete sieve.

			x - 0 1 2 3 4 5 6 7 8 ...
	$x \equiv 1$	(mod 3)	mod 3 - $\times_0$ $\surd_1$ $\times_2$ $\times_0$ $\surd_1$ $\times_2$ $\times_0$ $\surd_1$ $\times_2$ ...
(a)	$x \equiv 2,3$	(mod 5)	mod 5 - $\times_0$ $\times_1$ $\surd_2$ $\surd_3$ $\times_4$ $\times_0$ $\times_1$ $\surd_2$ $\surd_3$ ...
	$x \equiv 0,2,3$	(mod 7)	mod 7 - $\surd_0$ $\times_1$ $\surd_2$ $\surd_3$ $\times_4$ $\times_5$ $\times_6$ $\surd_0$ $\times_1$ ...
			standard ordering of residues
			x - 1 4 7 10 13 16 19 22 25 ...
	$x \equiv 1$	(mod 3)	mod 3 - $\surd_1$ $\surd_1$ $\surd_1$ $\surd_1$ $\surd_1$ $\surd_1$ $\surd_1$ $\surd_1$ $\surd_1$ ...
(b)	$x \equiv 2,3$	(mod 5)	mod 5 - $\times_1$ $\times_4$ $\surd_2$ $\times_0$ $\surd_3$ $\times_1$ $\times_4$ $\surd_2$ $\times_0$ ...
	$x \equiv 0,2,3$	(mod 7)	mod 7 - $\times_1$ $\times_4$ $\surd_0$ $\surd_3$ $\times_6$ $\surd_2$ $\times_5$ $\times_1$ $\times_4$ ...
			optimized ordering of residues

Figure 1-2. Single Residue Optimization

A dramatic increase in speed can also be realized at absolutely no cost by simply relabeling ring positions so that the sieve tests only values that satisfy any congruences with a single acceptable residue class. For example, if $x \equiv a \pmod{m}$ is specified by the problem, then ring i can be labelled so that consecutive positions differ by $m \pmod{m_i}$ rather than one; the sieve then skips over the $m-1$ values not congruent to $a \pmod{m}$ between tests, increasing performance by a factor of m. Figure 1-2 shows an example in

which this technique increases the rate of sieving by a factor of three. For each ring where $m_i \neq m$, the optimization simply permutes the standard arranement of residue classes within the ring. However, for a ring in which $m_i = m$, the optimization fills the ring with the single acceptable residue class, a, meaning the ring does not have to be tested at all.

For problems containing more than one single residue congruence, the optimization technique can be repeated using each congruence to increase the sieving rate by a factor equal to the product of the moduli involved. The easiest method of implementing such an approach is to combine the single residue congruences and use the resulting congruence as the basis for reordering the remaining congruences. Optimization can also be performed using a congruence of the form $x \equiv a_1, a_2, \ldots, a_r \pmod{m}$ by partitioning the original problem into r sub-problems involving a single residue classes modulo m. This produces an increase in overall sieving speed of $\frac{m}{r}$, but incurs the overhead of running r problems.

1.2.2.2 FILTERING SOLUTIONS

Automatic sieving systems are frequently required to run sieving problems that cannot be implemented by the system's hardware, either because the sieve's rings cannot hold all of the problem's congruences or because the problem specifies restrictions on its solutions in addition to those imposed by the congruences. Any such problem P can still be solved by partitioning it into two subproblems: P_r, the problem that defines only the congruences of P that can be loaded into the sieve's rings, and P_s, the problem that defines the remaining congruences and any additional restrictions. Since all solutions to P are also solutions to any problem P' formed by removing one or more congruences or additional restrictions from P, $S(P)$ can be found by taking the intersection of $S(P_r)$ and $S(P_s)$. In practice, it is not necessary to determine $S(P_s)$; instead, one can generate $S(P_r)$ using the sieve hardware and then test each solution individually to see if it also lies in $S(P_s)$. This latter step is termed *solution filtering*.

A sieving system can implement filtering in two ways. The simplest method is to have the system solve P_r in the normal fashion and then test the elements of $S(P_r)$ afterwards; the advantage of such *off-line* filtering is that it can be done on any automatic sieving system. Alternatively, the system can test each element of $S(P_r)$ as it is produced by the sieve hardware. While it requires more effort to implement, *on-line* filtering is essential if P_r generates a large number of solutions or if P is bounded by the number of solutions to be found rather than the end of a search interval.

1.2.3 Automatic Sieving Systems

From its first inception in the early part of the 20th century, the automatic sieving system has developed in an evolutionary, rather than revolutionary, manner. All of the systems built so far have been based on the basic sieve model described earlier, and differ only in the hardware used to implement the sieve and the amount of human intervention required to run the system. The various improvements that have been made have normally resulted from developments in technology, with a new sieving system being built whenever the hardware of the day could be adapted to produce a better sieve. Unfortunately, the relatively obscure nature of sieving has meant that the evolutionary process has tended to proceed in an erratic fashion; the few systems that have been constructed were usually the product of a few dedicated individuals working on a part-time basis with little or no support. Consequently, this account relates many failures and few total successes.

1.2.3.1 KRAITCHIK'S SIEVE

The first proposal for an automatic sieving system seems to have been made by Maurice Kraitchik [Kra22], who suggested building a sieve made up of gears in 1922. Fifteen gears representing congruences for the first 15 primes (or small multiples) would rotate freely along a common axle; each gear would have one tooth per residue class. A second set of 15 gears, each containing 64 teeth, would be fixed to a second axle mounted in parallel with the first. By connecting this axle to a motor, the latter gears would cause the former to advance at a uniform rate of n teeth per second. Kraitchik suggested placing electrical contacts at the gear positions corresponding to acceptable residues, allowing the machine to detect when all of the gears presented an acceptable residue simultaneously and stop automatically. Alternatively, holes could be drilled in the gears and a light source placed at one end of the machine; if the light arrived at the other end, a solution was present. In either case, the user would inspect a counter that recorded the number of positions the gears had advanced and manually calculate the solution found.

Kraitchik did not speculate on how fast his machine would operate, but it would undoubtably have been much faster than the best stencil or strip methods then available. Although his proposal discussed a number of the important details, it was only an outline, not a full-fledged design, and no such machine was ever constructed. D. H. Lehmer has called the design "impractical", questioning whether the available technology would have been capable of manufacturing the required gears and whether in fact the two sets of gears

would have meshed properly [Leh89]. Nevertheless, it remains the first significant attempt to automate the sieve process.

1.2.3.2 THE BICYCLE CHAIN SIEVE

The first workable sieve was constructed by Derrick Henry Lehmer[3] in 1926 at the Berkeley campus of the University of California [Leh28], [Leh80]. Lehmer was a freshman at the time, and decided to build a machine after becoming frustrated when the series of 20 foot paper strips he was using to do sieving kept becoming entangled.

The sieve's rings consisted of 19 bicycle chains, each forming a loop that was draped over a 10 tooth sprocket. The sprockets were attached to a common axle which was turned by an electric motor. A small pin (actually a chicken wire staple) was placed on each link corresponding to an acceptable residue. Whenever a pin reached the top of the loop, it lifted a small spring and broke an electrical contact. The contacts for the rings were wired in series and connected to a relay that cut the power to the motor if all of the contacts were lifted simultaneously; the machine then coasted to a stop. A 12 ring version of the device is outlined in Figure 1-3.

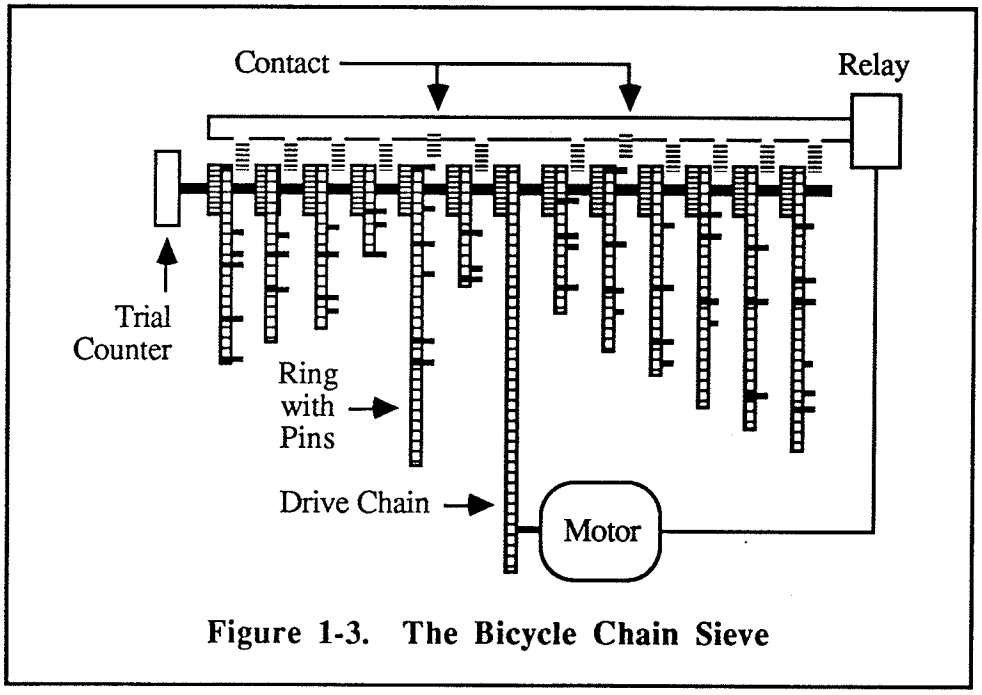


Figure 1-3. The Bicycle Chain Sieve

3 Son of Derrick Norman Lehmer, who developed the factor stencils.

Lehmer personally constructed the sieve in a few weeks, spending about $20 of his own money. The final product was capable of testing about 50 values per second. Although it was possible to go faster, the free hanging chains would start slipping once the sieve reached 60 trials per second. The soft wooden bearings used in the device also caused problems: they wore out rapidly and required constant tinkering. Even when the hardware was working, the sieve was not easy to use; setting up the rings was a cumbersome task requiring one to two hours, and when the machine found a solution and stopped, it remained stopped until its operator returned, backed the rings up to locate the solution, and restarted the search. Consequently, only one or two problems could be run in a week.

Despite these difficulties, the sieve was used on 50 to 100 problems over a period of about 8 months. Results included the factoring of previously intractible large numbers and the determination of some pseudo-squares [Leh28]. The machine was later disassembled by Lehmer and transported to Providence, R.I., but was stolen before it could be used again.

1.2.3.3 THE PHOTOELECTRIC SIEVE

The success of the bicycle chain sieve inspired Lehmer to attempt a more ambitious device in 1932 [Leh33b], [Leh34], [Leh80]. Drawing from Kraitchik's 1922 proposal, Lehmer made several important modifications to the design to make the concept workable. As before, the congruences were represented by n gears corresponding to the first n primes, with each gear having a different size and containing one tooth per residue class. However, the gears were mounted on individual axles and were advanced at a uniform rate by a set of identical gears mounted on a motorized axle. A set of holes corresponding to each tooth was drilled in each gear at a fixed distance from its circumference; each hole was plugged up with a piece of a toothpick if it represented a residue that was not acceptable for the problem being run. A beam of light shone through the holes at a fixed position; if it was detected by a photocell at the other end, a solution had been found and the machine stopped. An outline of a 12 ring version of the photoelectric sieve is shown in Figure 1-4; the actual device contained 30 rings. Note how a pair of prisms were used to fold the path of light beam; this made the machine more compact and allowed each section of the drive train to turn two rings instead of one.

Designed to run at 5000 trials per second, the new sieve was two orders of magnitude faster than the old one and considerably more complex. Its construction was a major endeavour lasting about 7 months and involving a variety of groups. Through the aid of his father, Lehmer was able to obtain a $1000 grant from the Carnegie Institution to build the machine. Lehmer had the Johnson Gearworks in West Berkeley put teeth onto the gear blanks, then drilled the residue holes himself. The other mechanical parts were made at Stanford University with the help of T. J. Palmateer. A three-stage photoelectric amplifier was built at Burt Scientific Laboratories in Pasadena; it amplifed the light signal by a factor of over 700 million, and was extremely difficult to construct. The complete sieve was assembled at the student lab at the University of California, but failed to work; a complete overhaul at Burt Labs was necessary before the bugs were finally worked out.

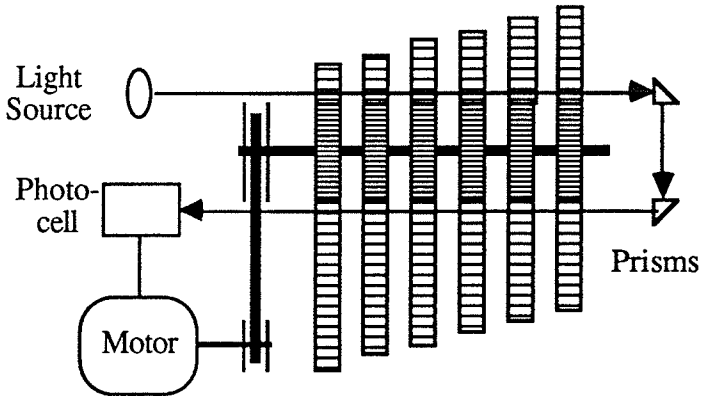


Figure 1-4. The Photoelectric Sieve

Like its predecessor, the photoelectric sieve was difficult to work with. Setting up a single congruence required the user to fill all the holes in each gear and then knock out the unneeded pieces of toothpick using dental instruments; creating a complete problem took over two hours. The rebuilt amplifier was so sensitive that it was set up in an adjacent room and heavily shielded with copper to prevent electrical interference from triggering the sieve to stop. Such precautions were not always enough: a nearby ham radio operator who normally broadcast between 10 and 11 P.M. often prevented the sieve from being used during that period.

During a two month period at Burt Labs, the sieve was used to aid in factoring numbers of the form $b^n \pm 1$; some results can be found in [Leh33c] and [BLSTW88]. The device was something of a sensation at the time, with glowing accounts of its design and

implications printed in newspapers [Kae33], [Oak33] and periodicals [Car33], [Leh33a]. Lehmer was soon asked to exhibit the sieve at the Century of Progress Exposition in Chicago; unemployed at the time, his initial reservations were overcome when the fair agreed to hire him as a demonstrator. Due to the nature of the demonstrations, no significant work was done while the machine was on display. Nor was it used in the years that followed. The formidable task of setting it up and using it apparently outweighed its great potential. An attempt to build a better amplifier was made at one point, but it was never finished. The machine languished in storage for a number of years before finding a permanent resting place at the ACM's Computer Museum in Boston.

1.2.3.4 THE MOVIE FILM SIEVE

The fate of the photoelectric sieve clearly demonstrated that the world's fastest sieve was useless if no one could be persuaded to use it. Consequently, ease of use became as important a part of sieve design as raw speed, and greatly influenced the design of Lehmer's next machine in 1936 [Leh80].

The new sieve was a simple variation of the first design, with bicycle chains replaced by loops of 16mm movie film leader. As before, the loops were draped over a common shaft, but a roller lubricated with talcum powder was placed at the bottom of each loop to ensure that the film holes remained firmly seated on the drive sprockets, since the weight of the film itself was insufficient for the purpose. A quarter inch hole was punched in a loop to indicate an unacceptable residue class. Solutions were detected by resting a metal screw on top of each film loop and applying an electric current; if the circuit to the axle was blocked by all 18 film loops simultaneously, a relay was tripped and the machine stopped.

Like the bicycle chain sieve, the movie film sieve was simple and cheap: Lehmer built the machine at his home in Pennsylvania over a two month period, spending about $50 for parts. Although it ran at only 50 trials per second, the sieve was quiet and required only about 30 minutes to set up a problem; consequently, it was frequently used. Since the film loops wore out after about ten hours of use, the sieve was normally put to work on small problems that required only one or two hours of running time.

When Lehmer returned to Berkeley he took the machine with him, where it was used until 1941. One of D. N. Lehmer's students, D. Swift, obtained a Ph.D. by using the sieve to run problems involving binary quadratic forms. It also examined values congruent to 1 and 3 (mod 4) to determine which class contained more primes. Unlike the two earlier

machines, the movie film sieve was never used for factoring or primality testing purposes. It is now on exhibit at the Computer Museum.

1.2.3.5 SOFTWARE SIEVES

The advent of digital computers in the latter half of the 1940's led Lehmer to experiment with the idea of programming a general purpose computer to do sieving. Not only was it far cheaper and easier to use existing hardware than to construct a specialized "one of" device, but the programmable nature of the machines made the task of switching from one problem to another a relatively simple task.

Lehmer's first attempt at sieving on a computer utilized the world's first all-electronic digital computer, ENIAC, over the July 4th weekend in 1946 ([Leh49], [Leh74]). Although the results of this work were used to factor certain numbers of the form $2^n \pm 1$ [Leh47], it was not really a software equivalent of the earlier sieves. Instead of solving the GSP, it found the smallest positive n that solved the single congruence $2^n \equiv 1 \pmod{p}$ by calculating $2^n \pmod{p}$ for $n = 1, 2, \ldots, 2000$. Sieving *was* utilized to generate the values of the primes p by eliminating composite values having at least one factor ≤ 47.

In the early 1950's, Lehmer programmed the Standards Western Automatic Computer (SWAC) to solve the GSP [Leh53]. Strings of bits were used to represent the congruences, with a "0" bit indicating an acceptable residue. The computer would merge sets of 36 residues from each ring and check for the occurrence of solutions, making the SWAC the first sieve to use multiple solution taps. A single test cycle required the machine to merge k 36 bit string segments in a serial fashion, check for solutions, and perform a circular shift on the k bit strings one by one; as a result, a large number of machine instructions were executed for every 36 values tested. However, since each instruction took very little time, the sieve program ran at about 1450 trials per second—about 30 times faster than the movie film sieve and almost 30% of the speed of the cumbersome photoelectric sieve. Lehmer used this speed to extend the pseudo-square search done using his first sieve [Leh54], as well as working on a variety of other problems.

In later years, sieving programs were developed for other computers. As available memory increased, sieving speed could be increased by combining sets of two or more congruences into a single larger congruence, thereby decreasing the number of bit strings to be merged. For example, John Brillhart got an IBM 7094 to sieve at a rate of 150 000 trials per second by compressing 21 or 22 congruences into 10 or 11 bit strings [BS67]; a

variety of new factorizations were discovered by this program using the difference of squares method [BLSTW88]. More recently, Michael Hermann and Cam Patterson benchmarked a 32 ring sieve on a Sun 4/280 at 2 000 000 trials per second [HP89].

1.2.3.6 THE DELAY LINE SIEVE

In an effort to reach higher sieving rates than were possible with conventional computers, Lehmer returned to the technique of building special purpose sieving machines. With the assistance of Paul Morton of the Electrical Engineering department at the University of California, Berkeley, he first attemped to construct a sieve out of switches and counters [Leh80]. Each counter cycled through a row of switches representing the residue classes for a congruence, routing the setting of each in turn to the solution detection circuitry. Thirty rows of switches provided congruences based on the first 30 primes, and utilized about 2000 switches in all. One of the major benefits of this approach was the ease with which a problem could be loaded. Unfortunately, the counter sieve never functioned reliably and was eventually abandoned after two years of effort.

Lehmer and Morton next built the Delay Line Sieve, DLS-127, in 1965. Although no formal description of the machine was ever given, aspects of its construction and operation are mentioned in [Leh66], [Leh68], and [Leh80]. The sieve had 31 rings, representing the primes from 2 to 127—hence its name. Each congruence was implemented by a segment of delay line, a form of wire that propogates an electrical pulse at a fixed speed. By forming a precisely measured loop and initializing it with a sequence of pulses representing residue classes, the pulses would circulate past a single solution tap and fed into a solution detection circuit. The delay line's high speed allowed the sieve to test one million values every second.

In addition to its record breaking speed, the DLS-127 displayed a number of important design innovations that greatly increased its versatility. Unlike the earlier hardware sieves, the task of setting up a problem on the DLS-127 was highly automated. A program running on an IBM 7094 allowed the sieve user to create a sieve problem based on any Diophantine equation $f(x,y) = 0$ by simply entering the coefficients of the polynomial, f; the resulting problem specification was punched into cards which were then transferred to paper tape. When one problem terminated, the next was read into the sieve from tape in a matter of seconds. Solution processing was also automated. Each time a solution was detected, the sieve shifted into "idle mode", printed the solution value, and resumed sieving

automatically; thus, the sieve could find multiple solutions to a problem without having to waste any time waiting for operator intervention. The problem of idling the sieve without losing the data it contained was accomplished by reconfiguring the delay lines into a single loop and continuing to circulate the pulses; to resume sieving, the device waited until the residues had returned to their original locations and then split the rings back into separate loops. The DLS-127 could also be run in "solution counting mode", in which it would simply count each solution found instead of stopping and printing its value. This mode was used when solving a problem with a high proportion of solutions to trial values, such as finding pseudo-squares.

The delay line sieve was undoubtably Lehmer's most successful hardware sieve. Run as an unsponsored educational project at the University of California's Berkeley campus, the hardware was built in less than a year at a cost of roughly $2000.[4] Since the basic sieving hardware contained no moving parts, the cost of operation and maintenance was negligible. The sieve was used on a wide variety of problems, including factoring ([BS67], [LL74], [BLS75], [LM78]) and a study of various properties of integer sequences ([LLS70], [Sha73]). After several years, the sieve was outfitted with six more rings (made out of shift registers rather than delay lines) and renamed the DLS-157. It was eventually retired in 1975 and can now be found in the Computer Museum.

1.2.3.7 THE ILLIAC IV SIEVE

In the early 1970's, Lehmer again utilized an existing computer to perform sieving [Leh76]. This time the basis was the ILLIAC IV, or I4, an experimental "one of a kind" parallel processor designed at the University of Illinois.

The I4 contained 64 processors operating in a single instruction, multiple data (SIMD) manner. As with the SWAC, the sieve operated by merging and shifting bits strings, but each congruence was implemented by a separate processor, allowing the sieve to shift its "pseudo-rings" in parallel as in the earlier hardware sieves. Testing for solutions was also performed in parallel by logically ORing sets of 64 residues from all 64 rings in a single instruction; a "0" bit in the result indicated a solution. This switch from serial to parallel operation, along with the speed of the I4 instruction set, allowed the sieve to test 15 million

---

4 The delay lines were obtained for about $150 after being rejected by the Navy for use in the tropics because they weren't protected against fungus!

values per second—over two orders of magnitude faster than previous software sieve implementations.

Since the experimental nature of the ILLIAC IV prevented Lehmer from utilizing it for sieving on a long term basis, the I4 sieve was only intended to be a demonstration of the feasibility of adapting a parallel processor to sieving. Despite its success, the lack of widespread availability of parallel processing systems hampered further developments along these lines. Instead, researchers experimented with hardware sieves that offered the potential for even higher levels of performance.

1.2.3.8 SHIFT REGISTER SIEVES

In 1962, Gerry Estrin and others suggested construct a sieving system capable of testing in excess of 100 million numbers per second by using readily available integrated circuits [CEFT62]. Each ring would consist of one or more high speed linear shift registers connected together to form a single circular shift register of the desired size. Using "1" bits to represent acceptable residue classes, the device would sieve by logically ANDing one bit from each ring and then shifting the bit strings. Utilizing chips capable of shifting every 200 nanoseconds, a t tap sieve would be capable of $5t \times 10^6$ trials per second. The proposal cleverly demonstrated that a shift register capable of shifting by one bit each clock cycle could be made to shift by t bits per cycle by relabelling the bit positions. It also recommended connecting the sieve to a dedicated general purpose computer that would implement in software any congruences that were not implemented in hardware, eliminating the need for operator intervention in such problems.

Curiously, Estrin's proposal was not acted on for over a decade, either by his group, or by Lehmer's, who opted to build the much slower delay line sieve instead. This may have been due in part to the radical nature of the project, which was far more ambitious than any previous hardware sieve, or it may have been too complicated and expensive to build using the integrated circuit technology of the time. By the early 1970's, however, the potential benefits of the shift register design eventually convinced Lehmer and Morton to build one [Leh80], [Leh89]. About the size of a breadbox, their sieve had only a single solution tap, but ran at 20 million trials per second; more taps were planned for later installation but were never added. A reference to the device as the "SRS-181" [LL74] indicates that it contained 42 rings, and, like the delay line sieve, it provided both solution counting and regular search modes. Instead of combining the sieve with a conventional computer, a special

board was constructed that would act as *host*: loading the problem into the sieve, monitoring its execution, and processing the answers that it generated. Unfortunately, before the board was completed, the sieve itself was mistakenly removed from its lab and sold as scrap while its owners were absent. A second model was never built. An even more ambitious shift register sieve was later planned by a group at the University of Illinois for use as a factoring machine [Don74]. This device was to have had 32 rings, achieving a sieving rate of 320 million trials per second using 32 taps. Although the project progressed at least as far as a partial set of schematics, it was apparently never built.

A complete system based on Estrin's model was finally constructed by Cam Patterson at the University of Manitoba in 1983 [Pat83], [PW83]. The University of Manitoba Sieve Unit, or UMSU,[5] contained 32 rings representing congruences for the first 32 primes. Eight sets of residues were tested every 60 nanoseconds, resulting in a sieving rate of just over 133 million trials per second. Unlike the SRS-181, it did not provide a solution counting mode, its designer deeming that small proportion of problems that would utilize it could not justify the extra hardware required. Altogether, the sieve required about 500 integrated circuits and three wire-wrap circuit boards, and cost approximately $7000 (Canadian).

Instead of having a dedicated computer act as its host, UMSU ran as a peripheral to an existing PDP-11/45 minicomputer. This not only saved money, but also allowed users to access the sieve from a number of terminals both on and off campus, rather than from a single location. Special software running on the PDP provided commands to create problems, queue them for execution, and inspect the results of completed problems. A permanent background process processed the entries in the problem queue in a serial fashion, automatically translating each into a sequence of commands that could be understood by UMSU's microprogrammed control sequencer, and processing the solutions it generated. Since most problems generated solutions infrequently, UMSU required little attention from the background process and had little impact on the PDP's other users.

The host software written for UMSU made the system far easier to use than earlier hardware sieves and was an important step forward in sieve design. Extending Estrin's suggestion, it allowed the user to include tests for any special solution requirements, as

---

5 Pronounced "ŭm sōō". The acronym was stolen from a more widely known UMSU on campus, the University of Manitoba Students' Union.

well as automatically simulating congruences that were not provided in hardware, making it the first hardware-based system to provide on-line solution filtering (see §1.2.2.2, "Filtering Solutions"). The software also handled the problem of hardware faults and power failures by verifying and recording the state of the current problem every hour, automatically restarting the problem after an error without losing more than the previous hour's work. On the other hand, the system did not attempt to optimize sieving when the problem contained one or more congruences with a single acceptable residue (see §1.2.2.1, "Optimizations"), nor could the user create and inspect sieving problems without temporarily suspending the execution of the current problem.

The UMSU system was used between 1983 and 1984 to find periodic continued fractions with long periods [PW85]. When the PDP minicomputer was upgraded to a Vax 750 in 1985, re-installing its UNIX-oriented software under the VMS operating system proved to be difficult; the different bus architectures used by the two minicomputers also forced modifications to be made to the parallel interface between UMSU and its host. A subset of the original software was eventually established, but the sieve developed a hardware fault shortly thereafter. Efforts to diagnose the fault were hindered by the departure of its designer and a lack of documentation, so the system was temporarily abandoned. In 1987, the arrival of a MicroVax II minicomputer running UNIX and utilizing a PDP-compatible bus encouraged a second effort to re-install UMSU. Patterson returned for a short period of time and was able to successfully repair the sieve and modify its software, so the complete system is again in operation. Subsequent research efforts have concentrated on finding polynomials that generate prime values [FW89], [MW89].

1.2.4 Summary

A variety of automatic sieving systems have been developed over the last 65 years, ranging from relatively slow and unwieldy mechanical devices to electronic computer systems that are over a million times faster (see Table 1-1). At the same time, significant (although less spectacular) gains have also been made in terms of system flexibility and the amount of human intervention required to solve a problem. Although considerable success has been achieved in adapting existing general purpose computers to carry out sieving, most of the major increases in sieving performance have resulted from the use of special purpose hardware that has been optimized for sieving. Apparently, the greater efficiency of a dedicated sieving machine can rarely (if ever) be matched using a general purpose computer built using comparable technology, and is well worth the considerable investment

of time and energy that its development entails. Dedicated sieving machines are also quite cost effective: they do not contain expensive hardware that is not used in sieving—such as a floating point arithmetic unit—nor do they have to be shared with other users. Even so, it is also apparent that great speed must be combined with "user friendliness" to ensure that a sieving system is successful. The examples of Lehmer's photoelectric sieve and the UMSU system illustrate that the world's fastest sieving machine can be rendered worthless if its users find it too difficult to work with.

Table 1-1.
Automatic Sieving Systems

Machine	Year	Rings	Trials/Sec
Bicycle Chains	1926	19	50
Photoelectric Gears	1932	30	5 000
16mm Movie Film	1936	18	50
SWAC†	1950's	?	1 450
IBM 7094†	1960's	21 or 22	150 000
DLS-127	1965	31	1 000 000
DLS-157	1969 ?	37	1 000 000
ILLIAC IV†	1970's	64	15 000 000
SRS-181	1975	42	20 000 000
UMSU	1983	32	133 000 000
Sun 4/280†	1989	32	2 000 000

† Denotes program running on a general purpose computer.

Chapter 2 :
The Open Architecture Sieve System

His reasons are as two grains of wheat,
hid in two bushels of chaff;
you shall seek all day ere you find them

William Shakespeare
The Merchant of Venice

The Open Architecture Sieve System (OASiS) is the immediate successor to the earlier UMSU system, and contains many significant improvements that make it the fastest and most flexible tool ever built for solving the generalized sieving problem. This chapter traces the development of OASiS, outlining the motivation behind its construction, the capabilities that it provides, and the major design decisions made in implementing them.

2.1 MOTIVATION

The construction of the UMSU system in 1983 gave the University of Manitoba's Department of Computer Science the fastest sieving system ever built, and one of its most successful research tools. However, experience with the system gradually exposed three significant shortcomings which limit its usefulness.

1) The sieve hardware supports only 32 congruence moduli, forcing the host system to implement all congruences of other sizes through software. If a problem has too few congruences in hardware, the host cannot keep up with the sieve, resulting in drastically reduced performance.
2) The host software makes no attempt to optimize sieving when congruences with a small number of acceptable residue classes are present, forcing the user to perform such optimizations manually. As a result, the system usually sieves at a small fraction of the speed of which it is capable.
3) The system is non-portable. The host software expects the host to run the UNIX operating system, and cannot be installed easily on other systems. Similarly, the interface between UMSU and its host requires the host to support an uncommon bus architecture.

Although it is possible to correct these problems by modifying the existing system, the task would be extremely difficult. UMSU's design is largely undocumented, and makes no provision for either hardware or software upgrades. As a result, even simple maintenance

is a major undertaking, and significant changes almost unthinkable. By May 1985, a desire to tackle problems that were beyond UMSU's capabilites, coupled with the difficulties encountered in installing the UMSU system on its new host, led Dr. Hugh C. Williams, its main user, to ask the author to build a totally new system as its successor.

2.2 GOALS

Setting goals for a sieving system is a difficult task because of the very nature of the generalized sieving problem. Sieving problems come in an unlimited number of shapes and sizes, making *any* limitation imposed by the system an arbitrary choice that restricts its ability to solve certain problems. As a result, the goals for OASiS are relatively vague, and tend to deal with qualitative issues rather than quantitative ones. Many of these goals were influenced by the presence of UMSU: those facilities that had worked well were kept, while those that worked poorly were improved. UMSU also provided a lower bound for the new system's sieving performance. In the end, five main points emerged as the basis for the new system.

1) *Generality.* The system would be a general purpose sieving system, able to solve any problem that UMSU could solve. In particular, the host software was required to provide on-line solution filtering to handle problem requirements that could not be implemented in hardware.
2) *High speed.* The sieve hardware would be capable of at least 200 million trials per second, making it at least 50% faster than UMSU. The system would also optimize sieving whenever congruences with a single acceptable residue class were present.
3) *Adjustable Rings.* Each of the sieve's rings would accept congruence moduli of different sizes, with the upper bound on size as large as possible to maximize the size and number of congruences that could be loaded.
4) *Reliability.* The system would be able to run for long periods unattended and recover from error conditions automatically without losing all previous work.
5) *Portability.* The system would be portable, and accommodate changes to its host computer (both hardware and software) easily.

In short, the new system would be the fastest sieving system ever built, with the ability to solve a wider variety of problems than could be done by UMSU, and able to withstand changes to its environment.

As work progressed on the new system, it was realized that its sieve could be designed with an "open box" architecture that would allow circuit boards to be added or removed at will. The advantages of this approach are numerous. For one, it would allow the sieve's capabilities to be increased by duplicating the existing hardware, or by building new hardware that is compatible with the old, allowing the sieve to be used on more complex problems or to take advantage of improvements in technology. The open architecture idea would also permit multiple sieves to be built to increase the rate of sieving beyond the limits of a single sieve. Even better, each processor could be equipped with as much hardware as required to solve the problem at hand; any spare hardware could then be given to the other processors rather than sitting idle. For these reasons, the following goal was added to the list of specifications.

6) *Flexibility*. The sieve would utilize an "open architecture", allowing it to accommodate enhancements readily, including the addition of extra processors.

In recognition of the potential provided by this design, the new sieve was christened the "Open Architecture Sieve", and the entire system, the "Open Architecture Sieve System".

These goals form the basis from which the detailed specifications for the new system were drawn up. The lack of concrete targets, such as the number of rings to be built, meant that many design limitations were set arbitrarily, and many were later changed as the design and construction process progressed. Implementation factors such as the cost, availability, and size of a particular part played an important role in shaping the system, and caused some proposed features to be scaled down, or eliminated entirely, in the name of practicality. The end product is the result of a balancing act between providing a system with too few resources to be useful and providing one with too many to be cost effective.

2.3 CHRONOLOGY

Work on OASiS began in May 1985, with research into the design of the sieve's hardware—in particular, into the design of the programmable size rings that are its main feature. Nearly a year was spent creating several alternative designs and in choosing the best one in terms of relative speed, complexity, and cost. The remainder of the sieve's hardware was then designed to complement the winning proposal. To test the feasibility of the basic approach, a prototype sieve with two rings was constructed out of breadboards in 1986. Tests on this device proved successful, and the decision was made to proceed with the development of the complete system.

In July 1987, work began on the full scale version of the OAS. The schematics for the sieve hardware were stored on a personal computer using OrCAD Systems Corporation's "SDT" schematic capture software; ACCEL Technologies' "Tango-PCB" and "Tango-Route" programs were then used to design the layout for the printed circuit boards. During both stages, the software proved its worth by detecting design errors and simplifying the process of making changes. Quite a number of design revisions were made, many of them occurring when multiprocessing features were added to the OAS design several months after work on the boards had been started. By year's end, the board designs were finalized and the required parts ordered. During the time the boards were being fabricated and the sieve's "off the shelf" parts were arriving from suppliers the OASiS software was designed and programmed. Assembly of the hardware and testing of the software proceeded in parallel, and was completed in September 1988. Subsequent testing exposed a number of major and minor problems that required changes to both hardware and software portions of the system. Finally, in January 1989, OASiS was put into regular use. In all, the system cost $10 600 (Canadian), of which $5600 was used to buy parts for the sieve; the remainder was spent to outfit the personal computer for printed circuit board design and construction.

2.4 DESIGN AND JUSTIFICATION

The Open Architecture Sieve System is a complex mixture of hardware and software. Both portions of the system provide a large number of features, all of which do certain things in a certain way. Since a complete list of system features would be quite lengthy, this section merely outlines the major capabilities of OASiS and their implementation, and attempts to justify their inclusion and form. Detailed accounts of these features and their operation are given in the ensuing chapters.

2.4.1 Design Philosophy

The design philosophy adopted for OASiS emphasizes simplicity and flexibility. A simple design was considered essential since changes to the original plan were inevitable and could be difficult to make if it was too complex. This is particularly true in the case of the high speed sieving hardware, where the designer's freedom to make modifications is restricted by the existing hardware and the laws of physics. If the hardware is too complex, a relatively minor problem can easily prove impossible to overcome. Simplicity

was also desirable for the OASiS software in order to minimize the effort involved in making changes and to maximize reliability. A flexible system design was necessary to achieve OASiS' goals of widespread applicability, portability, and openness, and was normally attained by avoiding design features that restricted its capabilities. In those instances in which such limitations were necessary, the system was often designed to support a number of alternative configurations, which allow the user to increase its capabilities if they are insufficient. Fortunately, since a simple system tends to be fairly flexible, the two aspects of the OASiS design philosophy tended to complement each other and were relatively easy to adhere to during the design and construction process.

2.4.2 Overall System Design

As with the UMSU system, OASiS utilizes both a conventional computer system and a specially constructed sieving device: in this case, a MicroVax II minicomputer system and the Open Architecture Sieve (OAS). The two are arranged in a master-slave relationship, with the computer taking the dominant role. This bipartite approach was used successfully in UMSU and breaks the system up into two relatively independent pieces, each of which takes upon itself the task(s) to which it is best suited. The sieve's role in the execution of a sieving problem is rather limited: it performs high speed sieving using some or all of the problem's congruences. In contrast, the host handles many things, including controlling the operation of the sieve, supplying any sieving capabilities that the OAS cannot, and handling the various aspects of problem management (problem creation and deletion, scheduling problems for execution, and the processing of problem results). This division of responsibility is consistent with OASiS' design philosophy since it moves the majority of the system's complexity into software, where it can be readily modified to enhance the system's capabilities.

The major tasks carried out by each portion of OASiS are outlined in Figure 2-1. As can be seen, the software running on the host computer can be partitioned into two classes: special software written for OASiS which controls the execution of a single problem, and existing system software that handles the problem before and after execution.

Open Architecture Sieve

- executes commands from the host
- does high speed sieving on a limited number of congruences
- notifies the host when a situation of interest arises
- performs almost no error-checking

OASiS Software Running on Host

- controls the execution of a single sieve problem
  - generates commands for the OAS to execute
  - records results of sieving in a file
  - optimizes sieving when single residue congruences are present
  - performs any on-line solution filtering required by the problem, including handling of congruences that do not fit into the OAS
  - performs extensive error checking of the OAS hardware, host software, and host-sieve communication
  - periodically verifies that the OAS is operating correctly and records a "checkpoint" if successful
  - allows the problem to resume from its last checkpoint after an error, or if the problem is terminated and later restarted
- allows inspection of a problem's results during its execution
- permits the user to terminate the current problem in an orderly manner

Host's System Software

- provides a file system for holding problem information
- provides a text editor for creating problems, examining results
- provides the job queue facilties used in problem scheduling
- automatically restarts the current problem after a system crash

Figure 2-1. Division of Labour in OASiS

2.4.3 The Open Architecture Sieve

The Open Architecture Sieve is a high speed sieving device based on an extremely flexible multiprocessor architecture. The sieve hardware can be configured to match the needs of the problem(s) at hand by altering the number of sieve processors used, the number of rings in each processor, and the size of each ring. At the present time, the sieve consists of a single processor with 16 rings, each of which can be loaded with any congruence whose modulus lies in the range 1 to 8192. The OAS provides a sieving rate

of nearly 215 million trials per second, and can be directed either to stop and report each solution it finds during sieving or to simply count solutions without stopping.

The most revolutionary feature of the OAS is its programmable size rings, which allow it to sieve any set of 16 or fewer congruences involving moduli no larger than 8K. Sets of more than 16 congruences can also be handled by combining two or more small congruences into a larger one; consequently, the OAS can be loaded with any set of congruences whose moduli involve the first 37 primes. Both the size and number of rings in the OAS reflect mathematical considerations (ie. size and number of congruences used in "typical" sieving problems) and implementation considerations (ie. total number of chips used, total cost, chip availability), and should be sufficient for many sieving problems. In the event that these limits prove too restrictive, the sieve can be upgraded to a maximum of 32 rings, and each ring increased to a capacity of 32K (32 768), with little difficulty.

The OAS was actually designed for a sieving rate of 256 million trials per second, to be attained using a 16 MHz clock and 16 solution taps. This was felt to be the highest rate that could be achieved without making the hardware too complex; a faster clock speed would have required a pipelined architecture or the use of ECL components, while more solution taps would have required many more chips. Although tests with a prototype indicated that the target rate was feasible, problems with the clock signal prevent the OAS from running reliably beyond 13.3 MHz, cutting the target rate by 17 per cent. However, experiments indicate that it may yet be possible to achieve 225 million trials per second or more.

The operation of the sieve is directed by its host, which translates the definition of a sieving problem into a sequence of low level steps that the sieve can execute and translates its responses into meaningful solution information. The commands available to the host allow it to load the sieve with a set of congruences, set the range of values to be searched, start and stop sieving, and inspect the solutions the sieve finds. A standard serial interface is used between the two machines to avoid the portability problems caused by UMSU's uncommon parallel interface. Serial transmission has the drawback of being slow, but since most sieving problems involve little communication between host and sieve, this is seldom important. Both host commands and sieve responses utilize the ASCII character set to aid in diagnosing system malfunctions: the user can either monitor transmissions by tapping into the communication line, or connect the sieve to a dumb terminal and enter commands manually. The latter is a simpler and more flexible means of troubleshooting than using pre-written diagnostic software, and prevents a bug in the host software from masquerading as a problem with the sieve.

Internally, the OAS is organized much like a conventional microcomputer system. Its brain is a simple microprocessor, which generates the control signals required to make the rest of the sieve hardware carry out the commands sent to it by the host. Abandoning the microprogrammed controller used in UMSU in favour of a microprocessor makes it easier to generate and modify the program code that controls the low level operation of the sieve hardware. It also provides extra flexibility, as was demonstrated when it was used to simulate solution counting hardware that had not been called for in the original sieve specifications. The loss of speed entailed in using a microprocessor is seldom important, since the microprocessor and the rings operate independently during sieving.

Physically, the sieve is composed of custom two layer printed circuit boards populated with commercially available high speed digital electronic components. This provides performance comparable to that of custom fabricated chips, but at a much lower cost. Only two types of board are utilized in the OAS, making the jobs of design, construction, and testing much easier than with previous sieves. The boards slide into a common backplane, and can be added or removed easily. Adding rings allows the sieve to handle larger sets of congruences, while the ability to remove rings without incapacitating the remainder of the sieve is useful if they became defective or are needed elsewhere. Installing additional sieve processors converts the OAS into a collection of independent sieves, which can be used co-operatively (to solve a single problem more rapidly by examining non-overlapping portions of the search interval in parallel) or individually (to solve several distinct problems simultaneously). In either case, the available rings can be distributed according to the needs of each sieve processor. Since repeatedly moving rings would increase wear on the circuit boards, the OAS is capable of merging two or more adjacent processors into a single unit, forming a larger sieve without requiring physical changes to the sieve hardware.

2.4.4 The OASiS Environment

The current configuration of the OASiS hardware is shown in Figure 2-2. Physically, the MicroVax views the Open Architecture Sieve as a dumb terminal. By signing on to the MicroVax's timesharing system and invoking the appropriate software, a user can utilize the sieve in much the same way as any other peripheral, such as a printer. The MicroVax can be accessed from a number of nearby video display terminals (VDTs) or from a remote computer through a local area network.

Access to OASiS is restricted to individuals with the username OASIS. This account has special privileges that are required for the OASiS software to function correctly and are not available to ordinary users. The current version of this software has been designed to exploit the existing sieve processor only, and ignores the OAS' multiprocessor capabilities. OASiS supplements the MicroVax's standard command set with a small number of new commands that allow the user to create new sieving problems, maintain a queue of problems awaiting execution, monitor or terminate the progress of the currently executing problem, and process the results of any completed problem. The host's other resources (compilers, printers, mail system, etc.) are not affected by OASiS and remain available for use at all times.

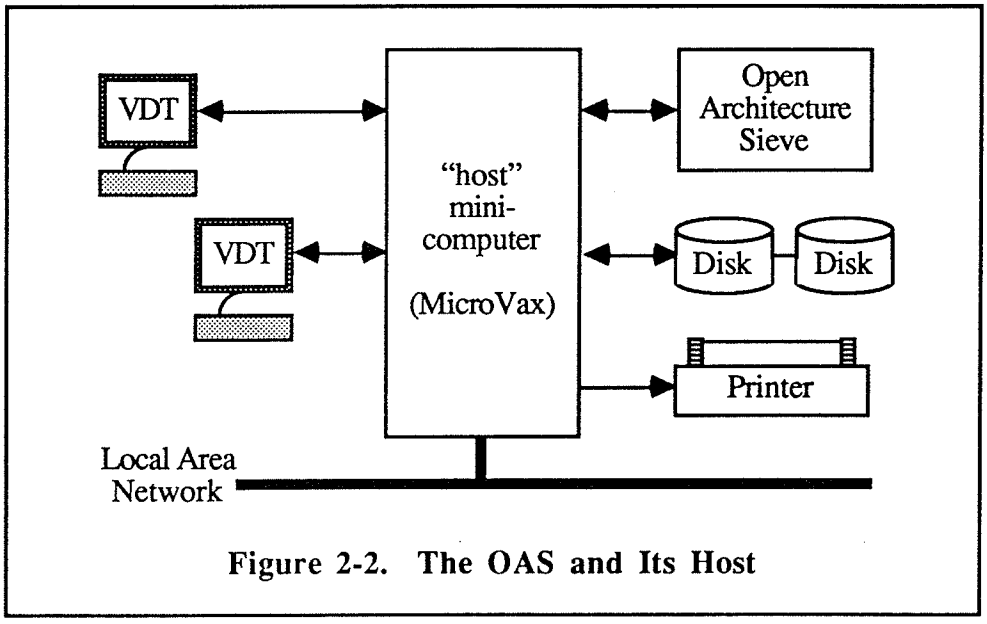


Figure 2-2. The OAS and Its Host

In OASiS, the user defines a sieving problem by creating from one to three files. A *problem file* is an ordinary text file that specifies the congruences used by the problem, the points at which sieving begins and ends, and the type of sieving to be done. Any additional restrictions defined by the problem require the creation of an executable file called a *filter program*, which accepts solution candidates and indicates whether or not each displays certain required properties. OASiS places almost no restrictions on a problem's congruences, search interval, or filter program, but the host software currently ignores the OAS' solution counting capability and only permits solutions to be recorded. While the problem file is a mandatory part of every OASiS problem, the filter program can be omitted if the problem imposes no additional restrictions. A second text file, called a *configuration file*, is also mandatory; it specifies what sieve hardware is available for use during sieving.

Since the current single-processor version of the OAS normally remains in the same configuration, OASiS uses a default configuration file if the user does not supply one.

The user can use any text editor provided by the host's system software to create problem and configuration files. Filter programs can be written in any language supported by the host, although C is preferable. If desired, the various files can be tested prior to execution to ensure that they contain no errors. When ready, a complete problem is then added to a priority queue of problems awaiting execution, after which the user may proceed with other work or sign off. Once a problem reaches the top of the queue, it is automatically executed and the results are appended to the problem file. For convenience, OASiS permits the user to view the problem file during execution to allow the progress of sieving to be judged. The problem executes until it is finished or until it is terminated by the user; in either case, the user is notified of completion through the host's mail system. Solutions can be extracted from a problem file at any point and either printed or used as input to some user written program that processes them.

During execution, the OASiS software performs extensive error checking on itself and on the OAS hardware; if a fault is detected, the software either restarts the problem or terminates it with an error message. A problem may also be restarted after a system crash (handled automatically) or after being prematurely terminated by the user (the user is required to resubmit the problem). To prevent large quantities of work from being lost in such cases, the software periodically takes a *checkpoint*; that is, it verifies the contents of the sieve hardware and records what point in the search interval has been reached. This allows a partially executed problem to be resumed from its most recent checkpoint, rather than starting over from the very beginning.

The capabilities of the OASiS environment allow the user to solve sieving problems quickly and easily, and are similar in most respects to those provided by UMSU. However, in terms of implementation, the systems' software differs in many respects. In line with its design philosophy, OASiS keeps almost all of the problem information and results in a single place—the problem file—where they can be readily inspected and processed; UMSU, on the other hand, stores the results of a problem separately from its definition and uses a machine readable format that is difficult for humans to comprehend. Similarly, the OASiS software is a collection of relatively independent, single function "tools", many of which are short, simple programs that can be easily written and maintained; as a result, OASiS' capabilities can be enhanced easily by modifying existing tools or adding new ones. In contrast, UMSU uses a pair of large, multi-function

programs that do not encourage changes. Some of the OASiS tools are actually standard commands provided by the MicroVax's system software. Incorporating such commands into OASiS allows users who are familiar with the MicroVax environment to learn to use OASiS quickly, and reduces the burden of writing and maintaining the OASiS software. Although this reliance on the host's system software might appear to make OASiS non-portable, it actually increases portability by encouraging the exploitation of a host computer's resources rather than specifying a rigid set of requirements that must be provided. Consequently, some form of OASiS could be established on almost any minicomputer or microcomputer system that provides a C language compiler.

2.5 PREVIEW OF SYSTEM DETAILS

The following four chapters describe the use and implementation of the Open Architecture Sieve System in detail. Chapter 3, "Using OASiS", is a user's guide to the system. It describes the various files used to define a problem and how the commands provided by the host software are used to solve it. In addition to acting as a reference for anyone wishing to use OASiS, Chapter 3 provides an introduction to the facilities of OASiS that is assumed during the later chapters on its implemenation. The following chapter, "OAS Principles of Operation" provides a similar introduction to the Open Architecture Sieve hardware and command set, describing the various components of the OAS, the interface between the OAS and its host, and the way in which the host can use the OAS to solve a sieving problem. The remaining two chapters provide a detailed account of the implementation of the software and hardware portions of OASiS. Chapter 5, "Host Software Implementation", discusses the internal operation of the various commands available to the OASiS user. Chapter 6, "OAS Implementation", covers the design and operation of the OAS hardware and the firmware that controls it.

Chapter 3 : Using OASiS

When all else fails—read the instructions.
Anonymous

The Open Architecture Sieve System allows its users to find solutions to almost any type of sieving problem quickly and easily. This chapter describes the facilities provided by OASiS for creating, executing, and terminating problems, as well as for processing their results and handling errors. The discussion assumes that the user is familiar with Digital Equipment Corporation's VAX/VMS operating environment, including its file system, logon procedure, and the basic commands provided by the Digital Command Language (DCL). A good introduction to these topics can be found in [Dig86]. It is further assumed that the user is signed on as username OASIS, since certain commands described below (denoted as "OASiS commands") are not available to other users.

3.1 CREATING A SIEVING PROBLEM

A sieving problem is created by constructing a problem file and, if necessary, a filter program or configuration file. To ensure that a problem has been defined correctly, OASiS allows the user to test the elements of a problem before submitting it for execution.

3.1.1 The Problem File

The problem file is an ordinary text file that defines the sieving problem to be solved. It may also contain directions about how the user wants OASiS to go about solving the problem. The information in the file falls into six categories.

1) The set of congruences to be used, giving the modulus and the acceptable residue classes for each. The moduli can range from 2 to 99 999, and there is no limit of the number of congruences that can be specified. The user can specify whether a congruence should be placed in the OAS or simulated by the host by forming a *congruence group.*
2) The points at which sieving begins and ends. Three different limits can be specified: the upper bound of the search interval, the maximum number of solutions to be found, and the maximum amount of time to be spent on the

problem. Any combination of limits can be specified, including none, making it possible to specify a problem that executes until terminated by the user.

3) The search mode to be used (ie. whether solutions are to be recorded in the problem file or merely counted).[1]
4) The name of the filter program to be used and the maximum time that can be spent testing each solution candidate. The filter program name is omitted if none is required.
5) The amount of time the OAS sieves before OASiS attempts to verify its contents. If no value is supplied, OASiS will take a checkpoint after every hour of sieving.
6) The number of non-fatal errors that OASiS accepts before aborting the problem. If no value is supplied, OASiS uses a default of three.

A complete description of the syntactic and semantic requirements for the problem file is given in Appendix E, "The OASiS Problem File".

Problem files can be created using any MicroVax text editor, or by writing a program which generates a text file as output. The latter method is normally worthwhile only if manually creating the files for a number of similar problems would be too much effort. By convention, problem files are always given a file extension of SIV.

3.1.2 The Filter Program

The filter program is an executable file that implements any restrictions on the solutions to a sieving problem above and beyond those imposed by the congruences specified in the associated problem file. During sieving, the OASiS software invokes this program to see whether a value that satisfies all of the problem's congruences is actually a solution to the problem. Many sieve problems do not make use of a filter program; those that do often require the same processing used in an earlier problem and for which an existing filter program can be employed. Thus, an OASiS user can go for long periods without having to create a new filter program.

The operation of a filter program is quite simple: it reads in solution candidates one by one from the "standard input" file, tests each value for a specified set of properties, and writes back an integer result to the "standard output" file. A result code of 0 indicates that

---

1 OASiS does not yet support the solution counting mode.

the candidate is acceptable, 1 indicates rejection, and –1 signals than an error occurred during testing. Although a relatively simple arithmetic calculation is usually sufficient, the filter program is free to perform any sort of computation it requires when testing a solution candidate, including reading and writing files, spawning subprocesses, communicating with other computers, and so on. The filter program terminates when end-of-file is encountered on the input file.

A filter program can be written in any language supported by the host, and must be compiled and linked into executable form before it can be used by OASiS. Since the solution candidates generated during sieving can often exceed the bounds of a standard 32 bit integer type, the filter program must be careful to avoid overflow errors. One way of doing this is to use the "mp" multi-precision integer package, which defines a data type that can hold integer values of any size and provides routines for doing arithmetic on integers of this type. (For more information on "mp", see [Ste89].)

To simplify the creation of filter programs, a generic mainline has been written which handles the mundane tasks of reading solution candidates and writing results; the user need only write a function called 'filter' which accepts a solution candidate and returns a 0, 1, or –1 result, as appropriate. Since the filter program mainline, and the "mp" package which it uses, are both written in C, it is recommended that the user's function also use this language. Such a routine would have the following form.

```
#include mp.h
/* any other "includes" that are necessary */

int   filter(value)
MINT  *value;
{
      int   res ;       /* filter result */
      /* body of filter routine */
      return(res) ;
}
```

The processing done by the filter routine body depends on the restrictions defined by the sieving problem; an example of a filter routine can be found in Appendix G, "A Sample Problem". After compiling the user's filter routine, it should be linked with the mainline code contained in [OASIS.PROBLEMS.FILTERS]MAIN.OBJ. The OASIS account has been set up so that the C compiler and the linker automatically include the files for the "mp" package, if needed.

A filter program can be tested and debugged using the MicroVax's normal debugging tools. The simplest method is to run the program interactively, manually entering test

values and observing the program's responses. End-of-file is indicated by typing CTL-Z. To test many values, the user can redefine the VMS logical names SYS$INPUT and SYS$OUTPUT to make the filter program read and write from files rather than the user's terminal. The generic mainline prints an error message every time the filter returns a –1 result. If desired, the user's 'filter' routine can have its own error message displayed by calling the multi-precise package's 'mp_genError' routine if it encounters an error condition. These error messages are intended as debugging aids only, and are ignored by the OASiS software during problem execution. When the user is satisfied that a filter program is fully debugged, it can then be used in a sieve problem without further changes.

3.1.3 The Configuration File

The configuration file is an ordinary text file that describes the current configuration of the OAS hardware. Its contents allow the OASiS software to communicate with the sieve and to utilize its resources properly during sieving. The items in the file fall into four categories.

1) The communication channel which the sieve is connected to.
2) The clock speed to be used during sieving.
3) The number of solution taps to be used.
4) The identification number and capacity of the available rings.

A complete description of the syntactic and semantic requirements for the configuration file is given in Appendix F, "The OASiS Configuration File".

A configuration file is normally created using an ordinary text editor and given a file extension of CON. Since the OAS hardware is seldom altered, it is rarely necessary for the user to create a new configuration file. The current OAS configuration is specified in the file [OASIS.SOFTWARE]SIEVE.CON. This file is used whenever the user does not specify a configuration file in an OASiS command.

3.1.4 Checking a Problem for Errors

A problem file and configuration file can be checked for errors prior to execution by invoking the OASiS *problem checker*. The checker is primarily concerned with detecting syntactic and semantic errors in the files, but also produces information on how the problem will be loaded into the OAS. The checker is invoked by entering the command

$ fit *probfile* [*configfile*]

where *probfile* and *configfile* are the problem file and configuration file to be used, respectively. The appropriate file extension (SIV or CON) is assumed if none is supplied by the user. If *configfile* is omitted, the default configuration file is used.

The FIT command first ensures that both files are syntactically and semantically valid. If an error is found, the offending file is listed at the user's terminal with an error message inserted at the point where the error was detected. If no errors are found, the command determines how the problem's congruences will be loaded into the sieve's rings and displays the following information.

1) A list of the rings and the congruences that are loaded into each of them. The combined modulus of each ring's congruence(s), as well as the percentage of the ring's capacity that it utilizes, are also displayed.
2) A list of the congruences that have only a single acceptable residue, as well as their combined modulus. Such congruences need not be examined during sieving, allowing OASiS to increase its normal sieving rate by a factor equal to the combined modulus.
3) A list of the congruences that are implemented by the OASiS software and which filter the solution candidates generated by the OAS.
4) A report on the "goodness" of the fitting, including the number of congruences that are handled in hardware and the portion of the rings' total capacity that is used.
5) Performance estimates, including estimates on the interval between solutions generated by the OAS and the time taken to load (or checkpoint) the problem. The latter figure is based on the number of characters transmitted to the OAS and the bandwidth of the interface between it and the MicroVax; however, since the two devices exchange data in a manner that does not exercise the interface to its full capacity, the loading (or verification) process actually takes about five times as long as the estimate shown.

The information provided by the FIT command allows the user to judge if a problem is well suited to the capabilities of OASiS. If too few congruences are loaded into the sieve, the MicroVax will be overwhelmed by solution candidates that must be tested against the problem's other congruences, thereby slowing the rate of sieving dramatically. If such a situation arises, the user can adjust the problem file to override the default mapping of congruences to rings performed by OASiS in an attempt to obtain a better fitting (see

§E.2.2.2, "Congruence Groups" in Appendix E). The modified problem can then be rechecked to determine the effectiveness of the user's changes.

3.2 RUNNING A SIEVING PROBLEM

Once a problem has been created and checked for errors, it is normally submitted for execution. OASiS maintains a queue of sieving problems, which are executed in sequence according to their relative priorities. During the execution of a given problem, the user can inspect the results that have been generated at any time and, if desired, terminate the problem in an orderly manner. After the problem has ended, its solutions can be isolated for further processing.

3.2.1 Submitting a Problem for Execution

A sieving problem is normally executed by placing a command file into the VMS batch job queue, SYS$OASIS. The MicroVax takes entries from this queue one at a time and executes the commands in the file, thereby invoking the OASiS software that performs sieving. When more that one job is available, the job with the highest priority is executed; if two jobs have the same priority, they are executed in the order they were submitted. Priorities can range from 1 to 100, with a higher value indicating a higher priority.

The user creates a command file for a sieving problem and submits it as a batch job by entering the following OASiS command.

$ qjob [*probfile*] [*configfile*] [*queuename*] [*priority*]

Here *probfile* and *configfile* are defined as before, *queuename* indicates the job queue the problem is added to, and *priority* indicates the problem's priority. If an argument is omitted,[2] the user is either prompted for its value (in the case of *probfile*) or given the choice of accepting or overriding a default value (in the case of the other arguments). By default, QJOB uses the default configuration file, the SYS$OASIS job queue, and a priority level of 50. The QJOB command creates a command file with the commands that initiate sieving and places it in the specified job queue. The command file is placed in the same directory as the problem file and uses a file extension of COM instead of SIV. The

---

2 Since the parameters to this command are positional, a null argument ("") must be supplied as a place-holder for each missing argument; this can be omitted if there are no non-null arguments which follow.

file should not be altered in any way until the problem has finished execution; it can then be deleted.

Once a job has been submitted, standard DCL commands allow the user to view the contents of the SYS$OASIS queue, alter a job's priority, or delete a problem prior to its execution. Commands are also available which allow the operation of the queue to be suspended and restarted, with options that either abort the current job or allow it to finish executing first.

3.2.2 What Happens During Problem Execution

Once invoked, the OASiS sieving software solves a sieving problem without need for user intervention. It automatically reads the definition of the problem contained in the problem file, loads the Open Architecture Sieve with as many congruences as it can, and begins sieving. Each time a solution candidate is generated by the OAS, it is tested against any congruences that were not loaded into the sieve and then against the filter program specified by the problem file (if any); any value that passes all of these tests is considered to be a solution. The software also takes a checkpoint at regular intervals to enable a partially executed problem to resume execution at the point where it left off (or close to it), rather than from the very beginning. Sieving continues until any of the termination conditions defined for the problem arises (ie. time limit, solution limit, end of search interval), or the user manually terminates the problem, or a fatal error condition occurs. Once the problem ends, the final commands in the COM file mail a message to username OASIS, notifying the recipient that the problem has ended.

The OASiS software maintains a log of all significant events arising during the execution of a sieving problem, including the solutions found, the location of checkpoints, and a variety of informatory and error messages. The execution log is located at the end of the problem file, and is the user's primary source of information about what has gone on during sieving and why it happened. It is also read by the OASiS software whenever a problem is restarted. A batch sieving problem also causes the MicroVax to create a *log file* in the same directory as the problem file (with a file extension of LOG). This file records all commands executed from the command file and the output they generate, and is of little interest to the user in most circumstances. However, if the sieve hardware malfunctions, the OASiS software may supplement its normal problem file error messages by sending more detailed information to SYS$ERROR, which, for batch jobs, is the log file. The log

file also gets the output generated by OASiS if the problem file or the configuration file contain errors, if it cannot write to the problem file, or if the sieving program itself crashes. The user can view the contents of the log file at any time during the execution of a problem, but should not attempt to alter it in any way until execution has finished, when it can be deleted safely.

Because sieving problems can execute for days or weeks, OASiS has been designed to handle a wide variety of hardware and software errors. The OASiS software performs numerous tests on its own operation and on its interaction with the Open Architecture Sieve; during its periodic checkpoints, it also checks that the congruences being sieved by the OAS have not been corrupted by a hardware "glitch". Errors which appear to be the result of random faults in the sieve hardware or the interface between the OAS and the MicroVax are considered to be *non-fatal*, and OASiS attempts to restart the problem from the most recent checkpoint. All other errors are considered *fatal* and cause the problem to be terminated. The occurrence of too many non-fatal errors (as defined in the problem file) is itself a fatal error.

The number of possible errors that can occur during the execution of a sieving problem are too numerous to list, but two deserve special mention: filter program malfunctions and system crashes. Since the filter program contains user-written code, OASiS considers it "untrustworthy" and requires that the user supply the maximum time interval that testing a solution candidate test is expected to take. Then, if the filter program enters an infinite loop or crashes while testing a solution candidate, OASiS generates a fatal error once the specified time limit has expired, rather than waiting forever for a response. If the MicroVax crashes during the execution of a batch sieving problem, the operating system automatically requeues the job on the SYS$OASIS queue, ahead of any entries with the same or lesser priority. When the problem is restarted, it resumes from its latest checkpoint.

3.2.3 Monitoring a Problem

Once a sieving problem begins execution, the DCL commands SHOW and MONITOR can be used to display the number of system resources (CPU time, I/O operations, etc.) the problem has consumed and its current rate of consumption, respectively. However, the user is usually more interested in determining how many solutions have been found, what their values are, and whether any errors have occurred during sieving. This involves looking at the contents of the problem file or, occasionally, the log file.

Unfortunately, the standard DCL command for examining a file

$ type *file*

displays the contents of *file* only up to the end-of-file marker that existed when the file was last opened. When used on the problem file for the currently executing problem, it lists the entire file up to the most recent checkpoint, but no subsequent results. This has the dual drawback of displaying information that the user has seen before (such as the problem definition itself) and omitting what has never been seen (the most recent results). Thus, the TYPE command is rarely used to examine a problem file.

The OASiS command

$ look *probfile* [*from*]

has been specially created for examining the problem file of an executing problem. It lists the contents of *probfile* from the first line that begins with the sequence of characters given by *from*; if *from* is omitted, the file is listed from the beginning. Thus,

$ look prob LOG

shows only the messages resulting from the execution of the problem file PROB.SIV, while

$ look prob " 2 10"

lists only the messages generated at least 2 days and 10 hours into its execution. Note that double quote characters are required if *from* contains whitespace characters, and that the number of blanks at the beginning of the line is significant. For convenience, the command does not distinguish between upper and lower case letters when matching *from* to the lines of *probfile*.

The LOOK command can also be used to display the last few lines of a problem's log file. Entering

$ look prob.log "$ show time"

lists the time the problem began execution and any error messages that have been written to the log file. The file can also be listed in its entirety using the TYPE command, but this displays the long list of commands that are executed prior to sieving and which are of little interest to most users.

3.2.4 Halting a Problem

It is occasionally necessary for the user to halt the execution of a problem manually—usually to allow the host minicomputer to be shut down for file backups, operating system

changes, or hardware maintenance. Entering the OASiS command

$ kill

prompts the user to specify whether the currently executing problem should be *terminated* or *aborted*, then ends the problem. If the problem is terminated, the OASiS software takes a checkpoint before the job ends; if it is aborted, no checkpoint is taken and any solutions found since the previous checkpoint are lost.

Aborting a problem does not always end execution immediately. The OASiS software only checks for the abort signal at certain times during its processing. If it is in the act of testing a solution candidate against the user's filter program, or taking a checkpoint, several minutes may elapse before the abort request is recognized and the problem ends; the exact interval depends on the filter program being used and the number of congruences contained in the OAS, respectively. To force a problem to end without any delay, DCL's STOP command can be used to kill the problem's batch job.

Once the currently executing problem ends, the next problem in the SYS$OASIS job queue normally begins execution. If the user is terminating a sieving problem to allow the system to be shut down and there are other sieving problems awaiting execution, the DCL command STOP/QUEUE/NEXT should be issued prior to the KILL command to deactivate the queue. The queue is automatically activated whenever the MicroVax is rebooted, but can also be manually activated using the START/QUEUE command. Since a problem that is manually killed disappears from the job queue once it finishes, should the user wish to resume the problem at a later time it must be manually resubmitted using the QJOB command.

3.2.5 Processing Problem Results

Once a sieving problem has finished executing, the user is free to examine and alter the contents of the problem file using an ordinary text editor. In addition, the OASiS command

$ pull *probfile* [*solnfile*]

can be used to extract the solutions from *probfile* and place them in *solnfile* where they can be processed more easily by user-written programs. Only verified solutions are copied, so problem files containing invalid solutions can be processed without difficulty; similarly, a solution appearing several times in a problem file appears only once in the output file. The command assumes a file extension of SOL for *solnfile* if none is supplied by the user. If *solnfile* is omitted entirely, the solutions are sent to SYS$OUTPUT (the user's terminal, by

default). If desired, the PULL command can also extract solutions from a problem file while the problem is executing.

3.2.6 Running a Short Problem

The user can run a short sieving problem without incurring the overhead of creating a batch job by entering the command

$ sieve *probfile* [*configfile*]

any time that the OAS is not being used; *probfile* and *configfile* are defined as with the FIT command. This invokes the OASiS sieving software interactively, and does not use a COM file, job queue, or LOG file. The software solves the problem exactly as described in §3.2.2, except that the SYS$ERROR messages go to the user's terminal by default instead of the log file.

When a problem is run interactively the user's terminal is locked for the duration of sieving, so problems requiring hours or days of execution time should always be run as batch jobs. Entering CTL-Y to interrupt the SIEVE command kills the problem abruptly, without any checkpoint or termination messages being written to the problem file. This means that the user must sign on to a second terminal in order to inspect the problem's performance or to terminate the problem.

3.3 HANDLING ERRORS DURING SIEVING

The OASiS software does extensive error checking during sieving and attempts to handle abnormal situations in an orderly fashion. Every error, either fatal or non-fatal, results in one or more messages being added to the problem file, so the user should never be presented with a situation in which things fail with no indication of the cause. The error messages generated by OASiS are given in plain English whenever possible and should normally provide enough information for an inexperienced user to determine the cause and correct it. Unfortunately, there are always situations in which the software can only report what has gone wrong and not why it went wrong, so some errors require a considerable understanding of the internal operation of the OASiS software, the MicroVax, and the OAS for the user to be able to diagnose the problem and then develop a solution.

In the event that OASiS does not appear to be functioning correctly, the user should first turn to the error messages provided by OASiS or by the MicroVax and attempt to

discover what has gone wrong. If a careful reading of these messages, together with any relevant documentation, does not provide an explanation, the user will have to gather more information. The first step is to see if the error is reproducible; if it is, the user should then determine whether the problem lies in the OAS itself or in the OASiS software that controls it, and to look more carefully at the appropriate portion of the system.

The OAS can be examined in one of two ways: either by connecting a terminal to the hardware and entering commands manually or by using the OASiS SEND facility to transmit commands via the MicroVax. The first method is fast and easy and only requires a knowledge of the firmware commands used by the sieve hardware; the second method requires learning about the SEND command as well, but allows the user to do things that are tedious or impossible to do manually. Neither method requires the user to write any programs. See Appendix C, "SIVMON Guide", for a description of the commands provided by the OAS, and Appendix D, "SEND Guide", for details on the SEND facility.

The OASiS software can be examined by inspecting its source code or by recompiling and relinking the code with the DEBUG option enabled and then rerunning the sieving problem using the VAX/VMS interactive debugger. The OASiS sieving software is written in C language, uses structured programming techniques, and is extensively annotated, so it should not prove too difficult to understand, despite its length. Unfortunately, the debugger may alter the operation of the program, making it difficult to find the source of the problem being investigated. For example, during the development of the sieving software, the author found that the program could not open the communication channel to the OAS hardware when the debugger was being used *unless* a breakpoint was inserted just prior to opening the channel; when the program was run without using the debugger there was no such problem. The reason for this difference remains a mystery.

In conclusion, OASiS has been designed to be robust enough to overcome minor errors automatically and to handle major ones in a manner that aids the user in determining the cause. While troubleshooting is never pleasant, the debugging facilties provided by the system should allow the user to begin the process with a reasonable foundation upon which to build. The author's own experience is that most problems with OASiS can be diagnosed using no other tools than the initial error messages and some careful thought.

Chapter 4 :
OAS Principles of Operation

They called aloud "Our sieve ain't big,
But we don't care a button! We don't care a fig!"
Edward Lear, *The Jumblies*

The Open Architecture Sieve is the most advanced sieving device ever constructed, and is the central component of OASiS. Capable of sieving at a rate of nearly 215 million trials per second using any set of 16 or fewer congruences with moduli up to 8192, the existing version of the OAS can solve a wider range of sieving problems than any previous sieve. As well, its "open architecture" design allows the OAS to be upgraded to handle more and larger congruences, or even turned into a multi-processor sieve, to expand its capabilities even further. This chapter describes the operation of the OAS from the host's viewpoint and explains how the sieve can be used to solve a sieving problem.

4.1 Design Fundamentals

The design of the Open Architecture Sieve allows it to be configured with one or more independent sieving devices called *sieve units*. Each sieve unit is a fully functional sieve containing the following components:

1) up to 32 programmable size rings, each holding up to 8K (or 32K) residues,
2) a 16 bit solution window,
3) a 16 bit solution mask,
4) a 48 bit trial counter,
5) an 8 bit clock control register,
6) a 64 bit solution counter.

When used correctly, these components allow a sieve unit to find or count the solutions to a set of linear congruences, to limit the size of the search interval being sieved, and to control both the speed of sieving and the points at which solutions are detected.

Each sieve unit in the OAS is controlled by a *host*, which uses its resources to solve a sieving problem by transmitting a sequence of commands to the sieve unit over a serial communication line. A sieve unit normally acts as a passive slave device to its host, waiting silently for a command and then executing it; the only time the sieve initiates communication is to signal the host whenever it stops sieving because it has found a

solution or reached the end of the problem's search interval. The commands recognized by a sieve unit's command interpreter are quite simple in nature, and allow the host to read and write the various sieve components, select the desired sieving mode, start and stop sieving, and determine the current operating status of the sieve unit.

The OAS currently consists of a single sieve unit with 16 rings, and thus can only work on a single sieving problem at a time. If it is configured with multiple sieve units, the host can use the OAS either to solve different sieving problems simultaneously or to solve a single problem more quickly than it could using a single sieve unit by allocating a portion of the search interval to each sieve unit. In either case, the sieve units would remain physically disjoint and do their sieving independently; thus, the remainder of this chapter can fully describe the operation of the OAS by concentrating on the case of a single sieve unit solving a single sieving problem. Some of the issues involved in having the host coordinate the efforts of multiple sieve units to solve a single problem are described in the final chapter of this thesis.

4.2 MODES OF OPERATION

An Open Architecture Sieve sieve unit is a modal device—that is, it provides different functions at different times, depending on its current state. The three main modes that the host can put a sieve unit into are *problem mode*, *idle mode*, and *sleeping mode*.

During problem mode, the sieve unit searches for solutions at high speed. The way in which solutions are processed is determined by the secondary *search mode* selected by the host: in *recording mode* the sieve unit stops sieving and notifies the host each time it finds a solution; in *counting mode* it counts the solutions it finds without stopping or notifying the host.

If the sieve unit is in idle mode, it does no sieving and allows the host to examine and alter its components or change the search mode. Thus, idle mode is used by the host to download problem information into the sieve unit or to upload its results after sieving. Since the components can only be accessed during this mode, it is also referred to as *I/O mode* in some OAS documents.

In sleeping mode, the sieve unit is deactivated and relinquishes control of its rings to an adjacent sieve unit, allowing two or more sieve units to be temporarily merged to form a larger sieve unit containing all of their rings. While the resulting sieve unit is still limited to

a maximum of 32 rings, the use of sleeping mode provides a more convenient means of altering the configuration of the OAS than making physical changes to its hardware. Once a sieve unit is placed in sleeping mode, it remains in a state of complete hibernation until instructed to "wake up", at which point it regains control of its rings and returns to idle mode. The operation of a sieve unit that has control of rings made available to it by another sieve unit is no different than that of one which has only its own rings.

The various combinations of OAS modes are summarized in Table 4.1. To solve a sieving problem, the host downloads the problem to the sieve unit while it is in idle mode, selects the desired search mode, and switches the sieve unit into problem mode. When the sieve unit signals the host that sieving has stopped, or a checkpoint is to be taken, the host switches it back to idle mode and uploads the relevant problem information. The host then returns the sieve unit to problem mode to continue sieving.

Table 4.1
OAS Mode Combinations

Main Mode	Search Mode	Sieving Action	I/O Permitted
Problem	Recording	Reports each solution	No
Problem	Counting	Counts solutions silently	No
Idle	—	None	Yes
Sleeping	—	None	No

4.3 BASIC SIEVING CAPABILITIES

The OAS sieve unit design provides room for up to 32 rings, numbered 0 to 31. Whether or not a sieve unit actually contains a ring with a given number depends on the configuration of its hardware and whether the adjacent sieve units have been put to sleep; any of the 2^{32} possible combinations is permitted. Currently, the OAS' sieve unit contains rings 0 through 15. Each ring can be loaded with the residues for any congruence with modulus m as long as $\frac{m}{\gcd(m,16)}$ does not exceed the ring's upper bound n, which may be either 8192 or 32 768 depending on the chips used to construct the ring. Since most sieving problems involve congruences with prime moduli, it is seldom possible to load a congruence where m is larger than n. To load a ring, the host specifies its number, the value of m, and the m residue classes; "1" bits are used to indicate acceptable residues and "0" bits the unacceptable ones. During sieving, the residues are tested by the sieve unit in the order in which they were loaded. Any ring that is not used in a particular sieving problem must be loaded with a "dummy" congruence for which all residue classes are acceptable.

An OAS sieve unit does its sieving in much the same way as any other sieve. When the host switches it into problem mode, the sieve hardware begins generating a sequence of high speed clock pulses. Each pulse processes 16 residues from each ring and then shifts the rings by 16 positions, with each ring acting like a cyclic shift register whose period equals the modulus of its congruence; the pulse also increments a 48 bit *trial counter*. Before the next clock pulse is generated, the sieve unit logically ANDs corresponding bits from each set of residues and places the results in a 16 bit *solution window*. If any solutions are found—as indicated by the presence of one or more "1" bits—the sieve unit stops generating clock pulses; otherwise, it continues sieving using the next 16 residues from each ring. The actions taken whenever solutions are found are dictated by the search mode that has been specified by the host. If solution recording mode is being used, the sieve simply notifies the host and waits; the host should then return the sieve to idle mode, read the solution window and trial counter, and determine the value of each solution found by making some simple calculations. If solution counting mode is enabled, the sieve automatically shifts into idle mode, increments a 64 bit *solution counter* by the number of solutions found, and resumes sieving without notifying the host.

Since many sieving problems place an upper bound on the search interval to be tested for solutions, the OAS has been designed so that a sieve unit will stop sieving and notify the host once the trial counter reaches its maximum value of $2^{48}-1$. So, by initializing the counter correctly before sieving begins, the host can program the sieve unit to search a given range of values and then halt. For example, to run an (unrealistic) problem which requires only a single clock cycle, the trial counter would be started at $2^{48}-2$. The host must always initialize the trial counter prior to sieving; if the problem has no upper bound, the counter can be started at zero. If a problem requires more than $2^{48}-1$ clock cycles to complete, the counter should be initialized to zero and then reset each time it reaches its maximum count—approximately every 250 days.

The host can start and stop sieving at will, but if the trial counter is already at its maximum value, switching from idle mode to problem mode does not cause any further sieving to be done. When switching from problem to idle mode, the sieve unit always completes its current clock cycle, so the host can generate its request asynchronously without ill effect. At any time, the host can query the sieve to determine its status, obtaining its current mode, the search mode currently enabled, the state of the solution window (solutions present or not), and the state of the trial counter (maximum count reached or not). The latter two pieces of information are particularly significant whenever

the sieve signals the host during problem mode since they allow the host to determine why the sieve stopped sieving without having to actually read the components involved.

4.4 ADDITIONAL CAPABILITIES

An OAS sieve unit provides a number of special features that are rarely used during normal sieving but which can be very helpful in debugging the unit's hardware. Two of the features take the form of sieve components, while the other two are commands that the host can issue.

The first component is a 16 bit *solution mask*, which is logically ANDed with the residues from the rings on every clock cycle during sieving. Thus, a "0" bit in the mask ensures that the corresponding bit of the solution window stays in the "0" or "no solution" state even if all of the rings have an acceptable residue in that position. Deactivating some of the sieve's solution taps can be used to isolate defective solution taps, while turning all of them off causes the sieve to run freely regardless of the operation of the rings. Normally, the host sets the solution mask to all "1" bits at the start of a sieving problem and leaves it alone from then on.

The second component allows the host to select from one of eight sieving rates by loading a *clock control register* with an integer from 3 to 10. The clock signal used by the sieve unit is generated by dividing the frequency of a fundamental clock signal by the contents of the register. The 40 MHz basis currently used by the OAS allows its host to run the sieve as fast as 13.3 MHz (215 million trials per second using 16 solution taps) or as slow as 4 MHz (64 million trials per second). Normally, the host specifies a value of 3 to maximize the sieve unit's performance, but slower clock speeds are often useful when trying to observe the operation of components with an oscilloscope or logic analyzer.

To provide the host with a finer degree of control over sieving than is possible using these two components, the sieve unit incorporates a *single step* command that runs the sieve in problem mode for a single clock cycle and then returns it to idle mode. This makes it very easy for the host to run the hardware at full speed over a short search range and then inspect whether the components are in their expected state. To simplify the process of checking the rings, a *ring verification* command is also provided; the host retransmits the residues for a ring (taking into account that they have been shifted from their original positions) and the sieve indicates if any mismatch is found. In addition to its use in

debugging a suspected hardware fault, the host can use this command during the normal execution of a sieving problem to periodically ensure that a hardware "glitch" has not corrupted the contents of any ring.

4.5 THE COMMAND INTERPRETER

The OAS command interpreter is a program known as SIVMON ("SIeVe MONitor"). The program is responsible for prompting the host for commands, executing the commands sent to it by manipulating the appropriate pieces of sieve hardware, and returning any results generated by a command. It also counts the solutions found during sieving (if the solution counting mode is enabled) and notifies the host whenever the sieve stops sieving automatically. A full description of the SIVMON program is given in Appendix C, "SIVMON Guide".

SIVMON uses the greater than character ('>') to prompt the host whenever it is ready for a command. Each command is a string of ASCII characters, beginning with a one- or two-character mnemonic (see Table 4.2) and continuing with zero or more alphabetic command options or numeric arguments given in hexadecimal form. The syntax of every command is rigidly defined and must be adhered to exactly—SIVMON does not even permit the host to correct typing errors. For safety, the command interpreter only accepts a command if the sieve is in a mode in which the command makes sense; for example, the STEP command cannot be issued if the sieve is already sieving. After a command has been executed, SIVMON prompts the host for another.

The host is allowed to control the output generated by SIVMON in two ways. It can temporarily suspend the generation of output by entering an XOFF character (CTL-S) and later resume it by entering an XON character (CTL-Q); this prevents the sieve from being able to flood the host with results faster than they can be processed. The host can also abort the execution of command at any point by entering a CANCEL character (CTL-C).

While the sieve unit is in problem mode, SIVMON continuously monitors the solution window and trial counter. If the counter has reached its maximum count, or if the window contains solutions *and* the search mode specifies solutions are to be recorded, the program signals the host that sieving has stopped by issuing an exclamation mark character ('!'). This character is repeated following every command prompt until the sieve is put into idle mode, but never interrupts a command after the first character has been entered by the host.

If SIVMON finds solutions in the solution window but the sieve is using counting mode, it does not signal the host but simply increments the solution counter appropriately and continues sieving.

Table 4.2
SIVMON Commands

Command Name	Mnemonic	Description
Count	C	Enable solution counting
Down	D	Enter sleeping mode
Find	F	Enable solution recording
Go	G	Start sieving
Halt	H	Stop sieving
Initialize	I	Reset sieve
Memory	M	Examine sieve controller memory
Query	Q	Return sieve status
Read Clock Control	RC	Read clock speed
Read Mask	RM	Read solution mask
Read Ring	RR	Read ring (has several options)
Read Solution Counter	RS	Read solution counter
Read Trial Counter	RT	Read trial counter
Read Window	RW	Read solution window
Step	S	Sieve for 1 clock cycle
Up	U	Awake from sleeping mode
Write Clock Control	WC	Load clock speed
Write Mask	WM	Load solution mask
Write Ring	WR	Load ring (has several options)
Write Solution Counter	WS	Load solution counter
Write Trial Counter	WT	Load trial counter
Write Window	WW	Clear solution window

4.6 SOLVING A SIEVING PROBLEM

The host can use an OAS sieve unit to solve a sieving problem by translating the definition of a problem into a sequence of SIVMON commands that cause the sieve to perform some or all of the necessary sieving; whatever cannot be handled by the sieve hardware must then be implemented by the host.

The first step in solving a problem is to determine how its congruences are to be loaded into the available rings. A sieve unit can often be made to run a problem in which the number of congruences exceeds the number of rings by having the host merge two or more congruences with small moduli into an equivalent congruence with a larger modulus. If it is impossible for the host to find a way of loading all of a problem's congruences into the sieve unit, the host should endeavour to load as many as it can.

Once the mapping of congruences into rings has been determined, the host should reset the sieve unit and then initialize all of its sieving components. This involves loading each available ring with a congruence, setting the trial counter to stop sieving after the required number of clock cycles, and loading the clock control register, the solution mask, and (if necessary) the solution counter with appropriate values. The host must also set the sieve unit's search mode to the required value. After initialization has been completed, the host can instruct the sieve to GO to begin sieving.

The host does not need to do anything while the sieve unit is sieving except wait for the exclamation mark signal indicating that sieving has stopped; once it appears, the host should HALT the sieve unit and determine the cause. If the solution window is not empty and solutions are being recorded, it is necessary to determine the value of the solution candidate(s) found and test them against any congruences that were not loaded into the sieve and also against any additional restrictions imposed by the problem. For maximum efficiency, the host should immediately restart the sieve after reading the solution window and trial counter and perform the necessary filtering while the sieve continues searching for more solutions. If the sieve stopped because the trial counter has reached its maximum value, the host should either reload the counter and resume sieving or terminate the problem, as appropriate; for a counting problem, termination would involve reading the solution counter.

If the host wishes to take periodic checkpoints during sieving, this can be accomplished easily. It first involves halting the sieve unit and, if solutions are being recorded, checking the solution window and processing its contents as described in the preceding paragraph. The host should then read the trial counter and calculate what point in the problem's search interval has been reached, after which it can verify each of the sieve unit's rings to ensure that they are in the expected state. The host can also examine the solution mask and clock control register to verify that their contents have not changed. If everything appears alright, the host can complete the checkpoint procedure by noting how far the search has gone and, if counting solutions, the number of solutions found.

When using an OAS sieve unit to count the number of solutions to a sieving problem, it is important to note the limitations of its solution counting mode. The first is that the mode is useless if even one of the problem's congruences cannot be loaded into the sieve, or if additional filtering of the solutions is required; such a problem must be handled by having the host verify each solution candidate found by the sieve and maintain the solution count itself. Since the time required to transmit a solution from the sieve to the host is enormous

when compared to the cycle time of the sieve hardware during sieving, the resulting reduction in sieving performance may make it impossible to traverse a large search interval in a reasonable amount of time. The second limitation is that the solution counting is performed by the SIVMON program which, when it detects that the sieve has found a solution, reads the solution window, restarts the sieve, and increments the solution counter. Since the program does not run as fast as the hardware, it is possible for many clock cycles to elapse before the counter has been incremented; if the rings generate another solution in that time, the sieve hardware is forced to wait. Thus, if a problem generates a large number of solutions, the rate of sieving may be slowed down to a small fraction of its normal pace, again making it impossible to solve the problem quickly enough. Finally, it should be pointed out that the sieve unit does not signal the host if the solution counter overflows; however, this limitation should not prove significant since the counter will not overflow for thousands of years if it is started at zero.

Chapter 5 :
Host Software Implementation

Errors, like straws, upon the surface flow,
He who would search for pearls must dive below

John Dryden, *All for Love (Prologue)*

The Open Architecture Sieve System provides a small set of commands that enable the user to create sieving problems, solve them, and process their results. This chapter describes some of the details of how these commands are implemented by a combination of standard MicroVax facilities and special OASiS software. At certain points it is assumed that the reader is familiar with details of the VAX/VMS operating environment. Thus, it may be necessary to consult the relevant MicroVax documentation from time to time to provide the necessary background for understanding the implementation of a particular feature of OASiS.

5.1 THE OASIS ACCOUNT

The OASIS account has been configured with a number of special features required by the OASiS software. It provides a small number of new commands and utilizes certain privileges that are unavailable to ordinary MicroVax users.

The file [OASIS]LOGIN.COM is automatically executed by the MicroVax whenever the OASIS account is activated. The commands contained in this file

1) establish the new OASiS commands FIT, QJOB, SIEVE, KILL, LOOK, PULL, and SEND,
2) set up default libraries for the MicroVax's C compiler and linker to simplify the creation of C language programs that use the "mp" multi-precision integer package, and
3) configure the user's process so that the user and a batch sieving job can exchange information through a shared VMS mailbox whenever a problem is to be KILLed.

It should be noted that this environment is used both by interactive users of OASiS and by the batch jobs created by the QJOB command. Thus, any facility that is available to the interactive user can be utilized by a batch job, and vice versa.

The OASIS account has the authority to exercise ALL privileges provided by VMS. Those that are actually required by its software include:

1) SYSPRV and PHY_IO, to enable the SIEVE and SEND commands to communicate with the Open Architecture Sieve over a standard terminal line,
2) DETACH, to permit the SIEVE command to invoke the user's filter program as a separate process,
3) TMPMBX, to allow the SIEVE and KILL commands to communicate through a temporary VMS mailbox, and
4) ALTPRI, to permit a batch job created by the QJOB command to alter its priority level when it is executed.

At the present time, all but TMPMBX are unavailable to normal MicroVax users, and must be explicitly granted when needed. The LOGIN.COM file establishes the privileges required by interactive users of OASiS; for batch users, the command file created by QJOB adds some additional privileges.

5.2 THE SIEVE COMMAND

The SIEVE command is responsible for the execution of a sieving problem, and is the cornerstone upon which the remainder of OASiS' software has been designed. It accepts the names of a problem file and a configuration file as its two command line arguments; if no configuration file is specified, [OASIS.SOFTWARE]SIEVE.CON is used by default. The command checks each file for errors; if it finds one, it writes a listing of the offending file and its error to the standard error file, SYS$ERROR, and then terminates. Otherwise, the command attempts to solve the sieving problem using the OAS, continuing until one of the predefined termination conditions is reached, a fatal error occurs, or the user manually terminates the problem. The SIEVE command normally appends the results of executing the problem to the problem file, but it may also write to SYS$ERROR in response to certain unusual error conditions.

By default, the SIEVE command adds the conventional SIV or CON file extension to the names of the problem file and configuration file, as appropriate; if either file uses a different file extension, it must be included as part of the command argument.

5.2.1 Implementation Overview

The SIEVE command is a C language program whose source code has been partitioned into over 20 different modules. The modules are relatively independent, having been designed using accepted software engineering techniques to minimize inter-module coupling and maximize intra-module cohesion. Thus, despite its considerable length (approximately 8500 lines of source code, including comments), the program is relatively easy to understand and modify, and operates reliably. All of the source code for the program is written in ANSI C using standard C language programming practices for structure and layout, and is extensively documented.

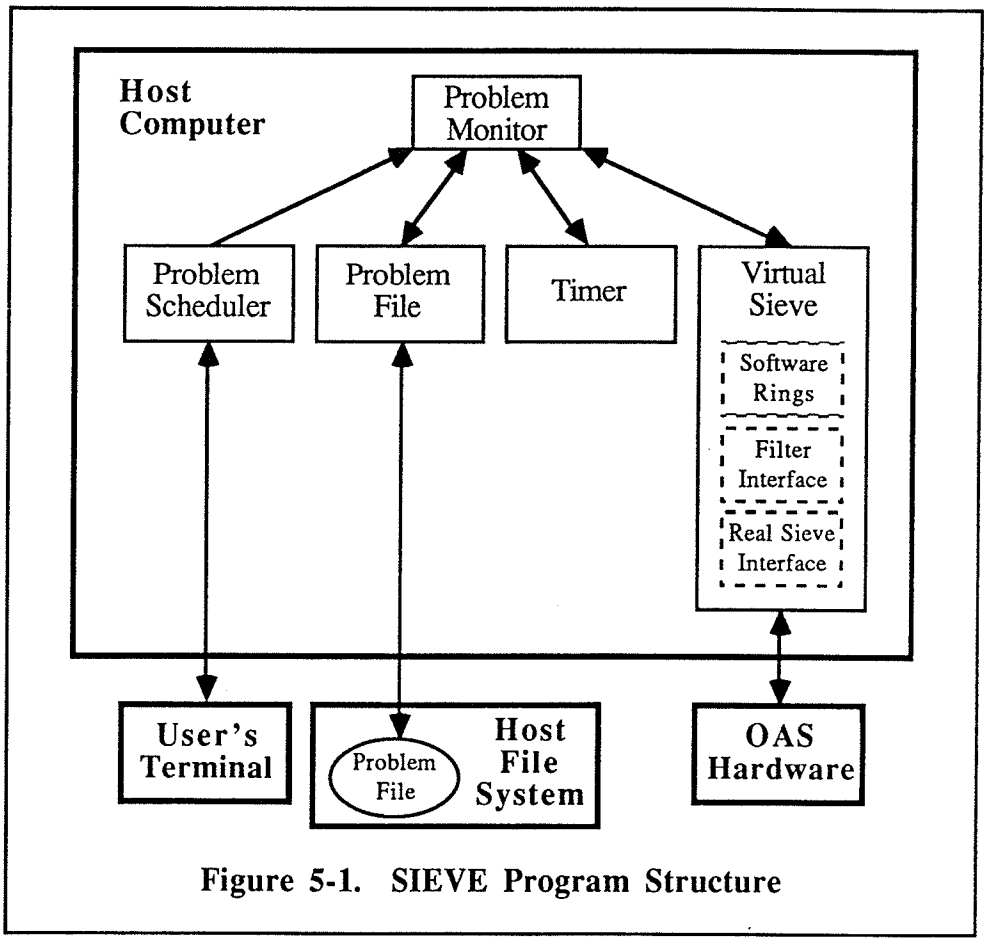


Figure 5-1. SIEVE Program Structure

The SIEVE program consists of an event-driven *monitor* that interacts with four distinct *device controllers* (see Figure 5-1). Each device controller is associated with a *device* (either real or simulated) which it watches constantly; whenever a predefined stimulus affects its device, the device controller adds an *event* to the monitor's *event queue*. The monitor processes events from the queue by directing one or more of the device controllers

to handle the stimulus; this may in turn generate further events. Since most devices can be stimulated in several ways, the device controllers are usually capable of generating more than one type of event.

The event-driven approach used in the SIEVE program was selected for a number of reasons. It is conceptually easy to understand, making the operation of the program easy to follow and predict. It also provides a natural means of partitioning the program into modules according to function. Finally, the design makes it easy to modify the program to carry out additional functions by adding new devices or new event types.

The following sections outline the operation and implementation of the monitor and its four device controllers.

5.2.2 The Monitor

The monitor oversees the execution of a sieving problem from beginning to end. It accepts the names of a sieving problem's problem and configuration files as arguments, initializes the four device controllers, and then enters an event processing loop. Events are taken from the event queue and processed in the order in which they are added, except for "user abort" events, which have a higher priority. The monitor exits the event loop and shuts down the four device controllers in an orderly fashion once it encounters an event indicating that the sieving problem has reached a termination condition specified in the problem file, has been terminated by the user, or an error has occurred. Finally, the monitor decides whether to continue executing the problem or not. If the event loop was terminated by a non-fatal error, the monitor resumes executing the problem from its last checkpoint; otherwise, the program ends.

To simplify the design and operation of the SIEVE program, and to increase its portability, a device controller is not permitted to interrupt the operation of the monitor in order to service its device whenever the device is stimulated. Instead, the monitor polls the controllers at the start of each iteration of the event loop, giving each controller the chance to check its device and generate any appropriate events. Initially, the monitor repeated such polling if the event queue was empty, but this resulted in the host computer spending most of its time checking devices that did not require any attention, and severely reduced the MicroVax's ability to service its other users. To overcome this problem, the device controllers now program their devices to signal the occurrence of an asynchronous stimulus

using a VMS *event flag*. Whenever there are no events to be processed, the SIEVE program hibernates until an event flag wakes it up; the monitor then polls the device controllers and processes any resulting events (or goes back to sleep if no events were generated). Since the new arrangement means that the SIEVE program utilizes the host's CPU only when it actually has something to do, and events occur relatively infrequently during most sieving problems, the MicroVax's other users are now rarely affected when OASiS executes a sieving problem.

The monitor is programmed to operate in a well-defined and predictable fashion, even though the events that it processes arrive in an unpredictable order, by acting as a deterministic finite state machine. This means that the type of processing performed in response to a given event is determined by the current state of the monitor and the type of event being handled, of which there are only a finite number of combinations. Unlike the case with a true finite state machine, the new state that the monitor enters after processing an event is not always determined by the initial monitor state and event type alone, but may be influenced by the results of the processing itself. This permits the number of states used by the monitor to be greatly reduced, and allows its source code to be written and understood more easily, by avoiding the introduction of states whose sole function is to check the results of the processing. The five states currently used by the monitor are given in Figure 5-2.

RESTART	- sieving problem has not yet started/resumed execution
SIEVING	- sieving problem is executing
DONE	- problem has reached search limit or solution limit; final checkpoint has been requested
KILLED	- problem has been manually terminated or run out of time; final checkpoint has been requested
OVER	- problem execution ended; monitor to exit after "cleaning up"

Figure 5-2. Monitor States

5.2.3 The Problem Scheduler

The problem scheduler controller is the portion of the SIEVE program that is responsible for scheduling of the execution of sieving problems; that is, passing problems

to the monitor in the order in which they are to be solved and terminating the currently executing problem at the user's request. However, in the current version of OASiS, the SIEVE program is invoked to solve a single problem, so no scheduling is required or implemented. As a result, the problem scheduler's role is limited to the generation of an event whenever the user requests the termination of the sieving problem. (A brief outline of how the scheduler could be extended to handle the full set of scheduling functions can be found in §5.6, "Porting OASiS to Other Hosts".)

Upon initialization, the controller creates a temporary VMS mailbox called SIEVE and initiates an asynchronous read operation that sets a VMS event flag when a message is placed into it. Each time the controller is polled, it examines the mailbox to see if a message has arrived; if one has, it generates a TERMINATE event if the message is a question mark ('?') or an ABORT event if the message is an exclamation mark ('!'). Since the generation of such an event eventually ends the execution of the problem, the scheduler does not bother to initiate a second read and ignores any further termination messages placed in the mailbox. When the controller is shut down, it deletes the mailbox it created.

5.2.4 The Problem File Controller

The problem file device controller is responsible for translating the problem definition contained in the problem file into an internal form that the SIEVE program can understand, and for adding the execution log messages generated during the execution of the problem to the end of the problem file.

When the controller is initialized, it parses the specified problem file using a hard-coded recursive descent parser that performs semantic as well as syntactic checks and returns the problem in an internal form. If the file contains an error, the parser automatically sends a listing of the file and the error to SYS$ERROR and returns nothing; otherwise, the controller opens the file for execution log messages and ensures that the problem has not previously reached one of its predefined termination conditions. If initialization is successful, the controller generates a LOAD_PROBLEM event that contains the problem information; if it fails for any reason, the controller generates a PF_ERROR event. The file remains open for log messages until the controller is shut down.

Unlike the other device controllers, the problem file device is never externally stimulated; thus, its controller is never polled by the monitor. Instead, the monitor passes it

information generated by the execution of the problem: solutions, checkpoints, and error and informatory messages, which the controller then appends to the problem file, along with the time it was generated. The controller normally leaves the file open for writing between messages, allowing other users to see (but not alter) the contents of the file; however, after recording each checkpoint, it quickly closes and reopens the file. This closure forces the MicroVax to update its file system directory to reflect the new end-of-file mark for the problem file, ensuring that all preceding output is preserved if the MicroVax goes down unexpectedly. The controller performs extensive error checking when writing to the file, and if it has trouble for any reason, it redirects all subsequent output to SYS$OUTPUT and generates a PF_ERROR event.

5.2.5 The Virtual Sieve

The virtual sieve controller is the most complex of the four device controllers, and solves a sieving problem by acting as a "virtual sieve" that contains an infinite number of rings and can perform any form of filtering. Since no such device actually exists, the controller utilizes the Open Architecture Sieve to solve as much of the problem as possible and simulates the missing capabilities itself.

When the controller is initialized, it parses the specified configuration file and attempts to establish a communication channel to the OAS. If an error occurs, a HARD_ERROR event is generated, representing a fatal error condition. Once the channel is opened, it remains open until the controller is shut down, thereby preventing other users (or other invocations of the SIEVE command) from interfering with the sieving being done by the current problem.

After a successful initialization, the monitor passes a sieving problem to the virtual sieve controller. The controller first sets aside any congruences that the user has indicated should not be loaded into the OAS, and also any congruences that contain only a single acceptable residue class; it then loads as many of the remaining congruences as it can into the OAS' rings—any remaining congruences are also set aside. The rings are loaded so that adjacent residue positions differ by the lowest common multiple of the moduli of the single residue congruences; thus, the OAS' sieving rate is effectively increased by a factor equal to the l.c.m. (see §1.2.2.1, "Optimizations"). The controller then instructs the OAS to begin sieving, and initiates an asynchronous read operation which sets a VMS event flag if the OAS requests attention from its host.

When the controller is polled by the monitor, it checks to see if the OAS has generated the exclamation mark character that indicates it has found a solution or the trial counter has overflowed. If so, it checks the sieve's status and takes appropriate action in response. If one or more solutions have been found, the controller reads the solution window and the trial counter, determines the values of the solution candidates, and tests them against any congruences that were not loaded into hardware and (if necessary) the user's filter program; for each value that passes these tests, the controller generates a SOLUTION event. If the solution limit is reached, or if the trial counter reaching its maximum count indicates that the end of the problem's search interval has been encountered, the controller generates a PROBLEM_DONE event; otherwise, the OAS is instructed to continue sieving. Because the virtual sieve controller often filters out all of the solution candidates found by the OAS, it may require attention but generate no events.

Periodically, the monitor instructs the controller to take a checkpoint, which always generates a VERIFIED or one or more SOFT_ERROR events. The controller then checks all aspects of the OAS hardware, spending the majority of the process in verifying the contents of the rings. Because the rings are most succeptible to errors, and the errors can be difficult to reproduce during debugging, the controller supplements its SOFT_ERROR event messages by sending a detailed failure report to SYS$ERROR whenever a ring malfunctions, which states exactly what the controller expected to see and what it found.

The controller performs extensive error checking during all phases of its operation, producing one or more SOFT_ERROR events if the OAS appears to have malfunctioned and one or more HARD_ERROR events if the user's filter program or the controller itself appears to be in error. All of the events generated by the controller are passed on to the problem file controller by the monitor for inclusion in the file's execution log.

This outline of the virtual sieve controller provides a framework for understanding its place in the SIEVE program, but omits a number of important details. For this reason the following sections have been included to elaborate on some of these issues.

5.2.5.1 LOADING THE OAS RINGS

One of the virtual sieve controller's most difficult tasks is to determine which of the problem's congruences are to be sieved by the OAS and which are to be handled by the controller itself. In order to utilize the OAS hardware to its maximum potential, the controller tries to assign as many congruences as possible to the available rings, often

combining two or more congruences and assigning them to the same ring. If two or more alternatives involve the same number of congruences, the one which can be transmitted to the sieve most quickly is considered best; for the OAS, this is the one in which the sum of the moduli of the congruences being loaded into the rings is smallest.

The task of finding the optimal mapping of congruences to rings is simplified by noting that if the congruences involved have moduli which are relatively prime (which is almost always the case), the best mapping always uses the congruences with the n smallest moduli. The proof is that any mapping of n congruences that does not include the smallest n moduli can be transformed into a mapping of n congruences that requires less time to transmit by replacing the congruence with the largest modulus by any unused congruence with a smaller modulus.

With this knowledge, the virtual sieve controller finds a mapping involving the maximum number of congruences by trying to fit all of the congruences into the rings, then all but the largest, all but the two largest, and so on, until a mapping is found that does not have any congruences left over. Unfortunately, the task of determining whether a set of congruences with relatively prime moduli fits into a set of rings of a fixed capacity is equivalent to the NP-complete problem of "multiprocessor scheduling" [GJ79], so the optimal mapping is difficult to compute. Currently, the virtual sieve controller simply tries three fast "bin packing" algorithms to see if any of them can fit the specified congruences into the rings, and then uses the best result. The "First Fit Decreasing" (FFD) approach processes the congruences in order of decreasing modulus and places each one in the first ring that it will fit into. The "First Fit Increasing" (FFI) algorithm operates like the FFD except that it processes the congruences in the opposite order. The "Best Fit Increasing" (BFI) approach places each in the ring that results in the smallest combined modulus. When a large number of congruences are involved the FFD approach tends to produce the best mapping, since it never requires more than 122% of the optimal number of bins. For small sets of congruences the BFI approach is usually best, since it produces a more even distribution of congruences than FFD (which fills some rings almost to capacity and leaves others empty), resulting in a smaller transmission time for the same set of congruences.

5.2.5.2 FILTERING SOLUTIONS

If a sieving problem specifies solution requirements beyond those imposed by its congruences, the virtual sieve controller handles the filtering of solution candidates by

passing each one to the user's filter program, which indicates whether or not the value is acceptable. The filter program runs as a separate process in parallel with the SIEVE process, and the two processes communicate by sending messages through a pair of VMS mailboxes.

When it loads a sieving problem, the controller invokes the filter program by creating a detached process, named OASIS_FILTER. The new process is assigned a run-time priority equal to that of the SIEVE program itself to ensure that it is neither starved for CPU time nor starves other processes. The controller also creates temporary mailboxes that the filter program uses as its standard input, output, and error files. After creating the new process, the controller ensures that everything is working correctly by testing the value 0; the result is ignored unless an error is detected. Following this check, the controller tests solution candidates only when they are generated by the OAS. The filter process and the mailboxes continue to exist until they are deleted by the controller when it is shut down by the monitor.

To test a solution candidate, the controller writes its value (as a character string) to the mailbox TOFILTER and reads its response from the mailbox FROMFILTER. When the filter's response appears, the controller generates a SOLUTION event if the candidate was accepted (0), discards it if it was rejected (1), or generates a HARD_ERROR if the filter program returned an error status (–1). Both the SIEVE process and the filter process hibernate while they are waiting for input from the other process, so they consume no CPU time unless they are actually performing useful work. To prevent the SIEVE process from waiting forever should the filter process enter an infinite loop or crash, the controller uses the value specified by the user to place a time limit on how long it will wait for a response after transmitting a solution candidate; if the time limit expires, the controller resumes processing and generates a HARD_ERROR.

5.2.5.3 INTERFACING TO THE OAS

The virtual sieve controller communicates with the OAS using a number of routines that collectively form the *real sieve interface*. The routines can be partitioned into three different classes: high level routines that perform a sieve operation by generating a sequence of SIVMON commands, mid-level routines that read and write integers and character strings, and low level routines that read and write single characters using the OAS communication protocol. The implementation of both the high level and mid-level routines is a fairly

straightforward decomposition into sequences of calls to lower level routines; the low level routines are more involved, and require calling VMS library routines to perform I/O over a MicroVax terminal line. The routines at all levels perform extensive error checking.

The most difficult part of the interface to implement is the set of low level routines which handle input from the OAS. An ordinary read operation has a three second time limit placed on it, after which the routine generates an error condition; this allows the SIEVE program to continue processing should the OAS malfunction and fail to respond as expected during a command. The asynchronous read operation has no such time limit; instead it programs the MicroVax to set an event flag when a character arrives, waking up the SIEVE program if it is hibernating.

5.2.6 The Timer

The timer controller manages the operation of a set of four programmable timers that can be used by the monitor and the other device controllers. Each timer can be set to generate a given event after a given time interval has elapsed. Currently, the monitor utilizes a maximum of two timers: one is set at the start of problem execution, and generates a TERMINATE event when the sieving problem has run for its allotted time interval (if any limit has been specified); the other is set once a checkpoint has been taken, and generates a BACKUP event when the next checkpoint is supposed to occur.

When the controller is initialized, it deactivates all of the available timers. When a timer is set, the monitor specifies the time limit to wait, the event to be generated, and a VMS event flag; the controller then instructs the MicroVax to set the specified flag after the specified interval has elapsed. Each time the controller is polled, it examines its set of timers to determine which (if any) have run down, generates the corresponding event for each, and deactivates its timer. When the controller is shut down, it does one last poll of the timers, deactivates them, and cancels any outstanding timer requests.

5.3 BATCH SIEVING PROBLEMS

Sieving problems are normally run as batch jobs that are executed in a serial fashion by OASiS from a queue of problems awaiting execution. Although the QJOB command has been written to handle the submission of a problem for execution, the majority of the batch job facilities are provided by the MicroVax's operating system.

5.3.1 The SYS$OASIS Job Queue

Sieving problems are normally placed in the VMS job queue SYS$OASIS, which has been created especially for this task. The queue executes its entries one at a time, and places no time limit on how long each entry can run. The standard queue manipulation commands provided by VMS can be applied to the queue to display and alter its entries, but can only be issued from the OASIS account—all other users are denied access to the queue by VMS.

5.3.2 The QJOB Command

The QJOB command submits a sieving problem for execution as a batch job, and is the only OASiS command that is implemented as a VMS command procedure instead of a C language program. It accepts four command line arguments: the problem file name, the configuration file name, the name of the job queue the problem is submitted to, and the priority level of the problem within the queue. If the problem file is omitted, QJOB prompts the user for it; if any of the others is omitted, QJOB asks the user to accept a default value or supply the missing value. The defaults used are the configuration file [OASIS]SIEVE.CON, the job queue SYS$OASIS, and a queue priority of 50. The QJOB command assumes the conventional SIV or CON file extension for the problem file and configuration file names unless a different file extension is specified by the user.

The QJOB command creates a batch sieving job in three phases: it verifies the command arguments, creates a command file that invokes the SIEVE command, and submits the file to the appropriate queue. In the first phase, QJOB ensures that the problem and configuration files exist and the priority level is an integer in the range 1 to 100; unfortunately, it cannot verify that the specified job queue exists, but if the queue is invalid, VMS generates an error message when the job is submitted. After argument verification, QJOB creates the command file for the sieving problem, and places it in the same directory as the problem file. It then submits the command file as a restartable batch job, and directs the job's log file to the same directory as the problem file. The restart option ensures that the problem automatically resumes execution when the MicroVax comes up after being shut down by the system manager or after a system crash.

5.3.3 The Batch Job Command File

The command file created by QJOB only issues a few commands during the execution of a batch sieving problem. First, it lowers the priority level of the batch job from 4 to 3 to ensure that the MicroVax's interactive users have higher priority than the sieve software. It then instructs the MicroVax to continue executing the command file if errors occur, so that the entire command file is always executed. Next, it displays the current time, allowing the user to tell when the batch job began and calculate how long the problem has been running by inspecting the job's LOG file during execution. The command file then issues the SIEVE command, which actually solves the sieving problem. Once the SIEVE command completes, the final commands in the command file mail a termination message to the OASIS account, indicating that the batch job has ended.

5.4 ADDITIONAL OASIS COMMANDS

The remaining commands provided by OASiS are relatively simple programs which perform single functions in a straightforward manner. Each program is written in ANSI C following standard C language programming practices for structure and layout. This section outlines the internal operation of the KILL, FIT, LOOK, and PULL commands; the SEND facility used to test the OAS hardware is described in Appendix D, "SEND Guide".

5.4.1 The KILL Command

The KILL command uses the standard input and output files, SYS$INPUT and SYS$OUTPUT, to prompt the user as to whether the currently executing sieve problem should be aborted or terminated and to obtain a response. If an abort is indicated, the program writes an exclamation mark ('!') to the VMS mailbox SIEVE, otherwise a question mark ('?') is written. Although the command does check that the mailbox write operation is successful, it does not wait for the character to be read by the active problem before finishing.

5.4.2 The FIT Command

The FIT command examines the specified problem and produces a report on how the problem is to be handled by OASiS when executed. The two command line arguments

used by FIT are the names of the problem file and configuration file, respectively; if no second argument is supplied, the configuration file [OASIS.SOFTWARE]SIEVE.CON is used. The problem file is checked for syntax and semantic errors; if an error is found, an annotated listing of the file is written to the standard error file, SYS$ERROR, and the command terminates. If the problem file is acceptable, FIT determines if the problem has been previously executed and, if so, whether it reached any of its user defined termination conditions; it then issues a message to the standard output file, SYS$OUTPUT, stating whether the problem is executable. Next, the configuration file is checked for errors, with the same sort of error handling as with the problem file if one is discovered. Finally, FIT generates a report on how the SIEVE command would process the problem, including a listing of how the congruences would be loaded into the OAS' rings and the amount of search optimization to be performed, along with some other figures that may be of interest to the user. This report is also directed to SYS$OUTPUT.

By default, the FIT command adds the conventional SIV or CON file extension to the names of the problem file and configuration file, as appropriate; if either file uses a different file extension, it must be included as part of the command argument. Unlike the LOOK and PULL commands, FIT cannot process the problem file for an active sieving problem; this is done to prevent the command's parse phase from failing if it encounters an execution log message that has only been partially written by the SIEVE program.

5.4.3 The LOOK Command

The LOOK command copies the specified input file to the standard output file, SYS$OUTPUT, starting at a specified line. The two command line arguments used by LOOK are the name of the input file and the characters making up the start of the specified line; if no second argument is supplied, the null string is used. The command searches the input file for a line that starts the same way as the argument string. The comparison is done on a character by character basis, without distinguishing between upper and lower case letters. A line is considered to match if all characters of the argument string match the corresponding characters of the input line. Once a matching line is found, it and all subsequent input lines are copied to the output file; when the command is done, the number of lines copied is written to the output file. The zero characters of a null argument string always match the first line of any input file, causing the entire file to be written as output.

By default, the LOOK command adds the SIV file extension to the name of its input file command argument; if the file uses a different file extension, it must be included as part of the file name command argument. The way in which the LOOK command opens its input file allows it to process the problem file and the log file for an active sieving problem.

5.4.4 The PULL Command

The PULL command copies solutions from the specified problem file to an output file. The name of each file is supplied as a command line argument; if no second argument is supplied, the command's output is directed to the standard output file, SYS$OUTPUT. The command reads the lines of the problem file from beginning to end. Each time a solution message is found in the execution log, it is written to a temporary work file; once a checkpoint message is read, all solutions in the temporary file are appended to the output file. When the command has finished processing the problem file, the number of solutions copied is written to SYS$OUTPUT.

Unlike the SIEVE and FIT commands, PULL does not parse the problem file—it merely inspects each line of the file to see if it has the correct form for a solution or checkpoint message. Consequently, the PULL command functions correctly only if each line of the execution log contains a single message. If the user alters the format of the problem file's log messages to deviate from this requirement, it may cause PULL to miss a solution or checkpoint. If the omission is significant, PULL will usually detect the situation when it recognizes a later solution or checkpoint and will issue an error message to SYS$OUTPUT in response; however, if the final checkpoint message in the problem file is missed, this recognition will not occur and valid solutions may be omitted from the output file. To avoid this sort of difficulty, it is recommended that the user not edit the problem file in any way once it has been executed by OASiS.

By default, the PULL command adds the conventional SIV file extension to the name of the problem file; if the file uses a different file extension, it must be included as part of the problem file command argument. The manner in which the PULL command opens its input file allows it to process the problem file for an active sieving problem.

5.5 INSTALLING THE OASIS SOFTWARE

All programs written for OASiS, with the exception of the LOGIN.COM and QJOB.COM files lying in the main [OASIS] directory, are located in subdirectories of the directory [OASIS.SOFTWARE]. Each subdirectory contains the source code for an OASiS command mainline or a module that is used in one or more commands. The SOFTWARE directory also contains a library of include files associated with these modules, a library for the object code of the modules, a "make" facility for rebuilding the OASiS software after modifications, and the default configuration file used by OASiS. The various files in the directory are listed in Figure 5-3.

Installing the OASiS software under VMS is a simple matter of copying the subdirectories and files of [OASIS.SOFTWARE] onto the host, along with the [OASIS]LOGIN.COM and [OASIS]QJOB.COM files. Typing

$ make [*debugflg*]

recompiles and relinks whatever portions of the OASiS software are missing or have been updated since the last time the software was rebuilt. The user can specify a *debugflg* value of DEBUG or NODEBUG. The former option builds the system to incorporate the MicroVax's interactive debugger. If the previous "make" did not use this option, all of the OASiS modules are recompiled and relinked. Similarly, the latter option builds the system without the debugger, rebuilding everything if the debugger was previously included. If *debugflg* is omitted, the system is built using whatever option was last used; the file in the SOFTWARE directory with the FLG extension allows the makefile to determine which option the preceding "make" used; it is updated by "make" after each run. None of this "making" is required to update or debug the QJOB command, since it is an interpreted DCL program rather than a compiled C language program.

5.6 PORTING OASIS TO OTHER HOSTS

In an ideal world, software written on one computer would run on others with little or no modification. Unfortunately, this is rarely the case for nontrivial programs. So, even though the OASiS software has been designed with an eye towards portability, it will almost certainly require significant changes if it is installed on another host computer. In some instances, it may not be possible to implement all of the features currently provided.

BIT.DIR	Variable-length bit string type package
BOOLEAN.DIR	Boolean type definition
CHECKER.DIR	FIT command mainline
CONG.DIR	Congruence type package
EVENT_FLG.DIR	VAX/VMS event flag definitions
EVENT_Q.DIR	Event queue type package
FILTER.DIR	Routines for interfacing with filter program
INT.DIR	General purpose integer routines
KILLER.DIR	KILL command mainline
LOOKER.DIR	LOOK command mainline
MIXER.DIR	Routines for mapping congruences into rings
MONITOR.DIR	Routines implementing OASiS monitor device (SIEVE command mainline)
MP.DIR	Multi-precision integer package
PARSER.DIR	Routines for parsing problem file and configuration file
PROB_FILE.DIR	Routines implementing problem file device
PROBLEM.DIR	Sieving problem type package
PULLER.DIR	PULL command mainline
REAL_SIEVE.DIR	Routines handling interface to OAS
RING.DIR	Ring type package
SCHEDULER.DIR	Routines implementing problem scheduler device
SCRIPT.DIR	SEND command mainline and sample scripts
TERMINAL_IO.DIR	Routines for communicating over MicroVax terminal line
TIMER.DIR	Routines implementing timer device
VIRT_SIEVE.DIR	Routines implementing virtual sieve device
HFILES.TLB	Source code library for include files
SIEVE.OLB	Object code library for non-mainline modules
1MAKEFILE.	OASiS software makefile
[NO]DEBUG.FLG	Flag file used during "make"
SIEVE.CON	Default OAS configuration file

Figure 5-3. Contents of [OASIS.SOFTWARE] directory

The only part of the OASiS software that is absolutely necessary is the SIEVE command; all of the other commands are merely conveniences that make the user's life easier. The basic requirements of the SIEVE program require it to read and write text files and communicate through a serial port, so it should be possible to make the program run on

any general purpose computer system that provides an ANSI C compiler. The major areas likely to need modification are those that utilize the host's file system or other machine specific features, including the problem file device controller, the timer device controller, the interface to the serial port, and the filter program interface. In most instances, this would involve replacing an existing call to a host system routine with an equivalent call to the new host's system software. The most difficult part of the program to modify would be the filter program interface, since some smaller computers do not allow programs to create a subprocess and communicate with it. In this case, the user's filter routine would be implemented as a subroutine and linked into the SIEVE program itself. If dynamic linking cannot be done by the program during execution, it would be necessary to build (and keep track of) a number of nearly identical versions of the SIEVE program—one for every filter routine used.

Once a working version of the SIEVE command was in place, it would be up to the user to decide which of the remaining OASiS capabilities to provide, and what form each should take. This decision is largely dictated by the nature of the new host system. In a multi-user system, such as UNIX, most of the capabilities could be implemented in much the same way as they are currently provided. On the other hard, a single user system like Apple's Macintosh would not easily support a second, independent process running on the computer at the same time as the SIEVE program. Thus, the current means of scheduling problems for execution, inspecting their results during execution, and manually terminating execution would have to be replaced. One alternative would be to have the SIEVE program provide these capabilities itself by altering the behavior of existing device controllers, or adding new ones, to handle new types of events. For example, the problem scheduler device controller could be modified to maintain a queue of sieving problems, which it would pass to the program monitor as requested. A new "user console" device controller could be set up to accept commands typed by the user, which would be passed on to the appropriate device controller for processing. Such commands could allow the user to alter the queue of problems to be run (which would be handled by a modified problem scheduler), terminate the execution of the current problem or the SIEVE program itself (which is already handled by the monitor based on request from the problem scheduler), and inspect the problem file for the current problem (which would be handled by a modified problem file controller). Such changes would still leave the SIEVE program's modular design relatively intact, allowing additional capabilities to be incorporated into the program in the future with the same relative ease.

Chapter 6 : OAS Implementation

I open with a clock striking,
to beget an awful attention in the audience
Richard Brinsley Sheridan, *ii*

The Open Architecture Sieve is essentially a special purpose microcomputer whose hardware and programming are dedicated to high speed sieving. This chapter describes the low level details of its design, constuction, and operation. A complete description of the OAS hardware can be found in Appendix A ("OAS Schematics") and Appendix B ("OAS PAL Descriptions").

Most of the components used in the OAS are available from a variety of manufacturers. For this reason, the chips mentioned below are usually designated by function, with a generic part number following in parentheses; when an uncommon chip is referenced, the manufacturer's name is also included. In the case of common electrical parts (such as resistors and capacitors), the part is designated by function alone. All of the major signals used in the OAS have names, which appear in upper case letters. Control signals are suffixed with (L) or (H) to indicate whether the signal is asserted by a LOW (0V or GND) or HIGH (+5V) logic level, respectively.

6.1 HARDWARE OVERVIEW

At first, the apparent complexity of the Open Architecture Sieve can be quite intimidating. In fact, the design is largely based on a few basic principles, and its components organized in a simple and regular fashion. This section outlines the overall construction of the OAS and the sieve units within it.

6.1.1 OAS Construction

The OAS is composed of custom two layer printed circuit boards stacked horizontally in a rack-mount card cage (see Figure 6-1). The vertical orientation of the boards allows convection to provide sufficient airflow to cool the sieve components without the aid of fans. The sieve is powered by a 35A power supply providing +5V and ground levels.

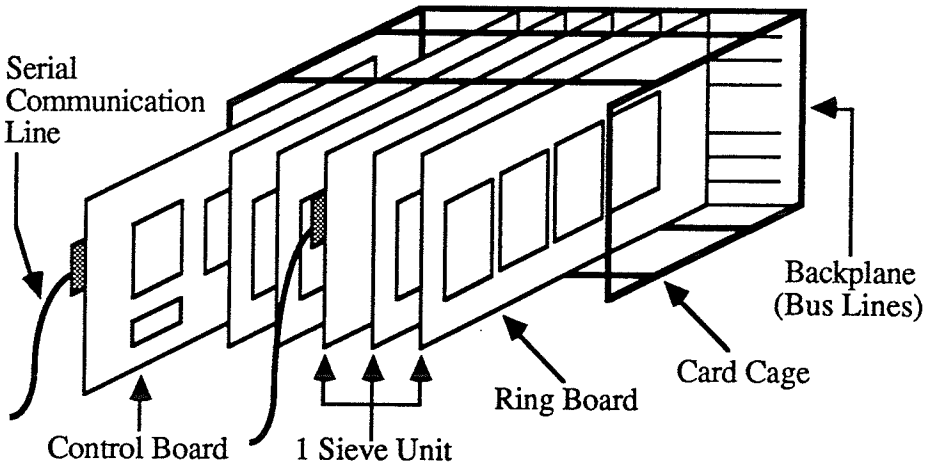


Figure 6-1. Open Architecture Sieve Construction

The OAS uses three different types of circuit boards: *control boards*, *ring boards*, and a *backplane*. Control boards and ring boards can be inserted in any of the cage's 13 odd-numbered slots, while the backplane is permanently mounted at the back of the cage. The backplane does not provide connections to its 13 even-numbered slots to prevent boards from being too close to one another and overheating. The card cage can be configured with one or more *sieve units*, each consisting of a single control board and the ring boards in adjacent higher numbered slots up to the next active control board or empty slot. As each ring board contains four rings, a sieve unit can use up to eight ring boards before its limit of 32 rings is reached. Each sieve unit in the cage communicates with its host over a separate serial communication line attached to its control board.

A sieve unit's control board communicates with its ring boards over the bus lines in the backplane using an architecture that allows the OAS hardware to be configured in a variety of ways. This lets the sieve user alter the number of sieve units and the distribution of rings between them to match the available boards to a specific sieving problem in the best possible way. Reconfiguration can also be done by having the host put a control board into sleeping mode; the board then logically disconnects itself from the backplane and its rings become part of the adjacent sieve unit. Any number of sieve units can be merged in this fashion, as long as the resulting sieve unit contains no more than 32 rings. While merging sieve units is easier on the hardware (and the user) than physically moving boards, it is less general; consequently, board swapping may be required to achieve a certain configuration.

To allow a sieve unit's control board to distinguish between its various ring boards, the user must assign identification numbers to the rings using a three position DIP switch located at the bottom of each ring board. The switches represent the most significant bits of the rings' ID numbers, with a closed switch representing a 0. Thus, closing all switches indicates that the board contains rings 0 through 3, while opening all switches specifies rings 28 through 31. Only one board in a sieve unit should be given a particular switch setting. Using duplicate settings does not damage the boards, but if multiple rings have the same number only the ring closest to the control board can be accessed by the host. Since the other ring(s) sharing that number cannot be initialized, the results obtained from sieving in such a situation will be unpredictable and almost certainly incorrect.

Currently, the OAS consists of a single control board and four ring boards, providing a single sieve unit with 16 rings. Ths sieve communicates with its host MicroVax II mini-computer over a 9600 baud terminal line. A second control board has been built and tested, but is of little use until enough ring boards have been constructed to complete a second sieve unit.

Building the OAS using two layer circuit boards rather than four layer boards with dedicated power and ground planes helped to minimize the costs of constructing the sieve, but also made it much more difficult to get the sieve to function properly. It was originally thought that a power and ground grid on opposite sides of each two layer board would provide enough current carrying capacity to run the sieve's many components, but these projections were not borne out in practice, resulting in severe problems with the operation of the ring boards. When the counter chips for certain rings were clocked, they drew more power than the traces running to them could handle, briefly lowering the voltage seen by the counters below their minimum acceptable limit by raising the ground level. As a result, these counters entered an unstable operating state and the rings in which they lay became desynchronized from the rest of the sieve. The problem was eventually solved by increasing the ring boards' capacity to sink power through a few additional ground wires, but many frustrating weeks were spent examining and modifying the OAS hardware before the problem of "ground bounce" was discovered and corrected.

6.1.2 Sieve Unit Design

A sieve unit is essentially a microcomputer that is specially designed for sieving. It contains a microprocessor, a small amount of memory, and a serial I/O port, but, unlike a

general-purpose microcomputer, it has no disk drives, parallel printer port, keyboard, or video output. Instead, it contains specialized sieving hardware, such as programmable size rings, a trial counter, and a solution window.

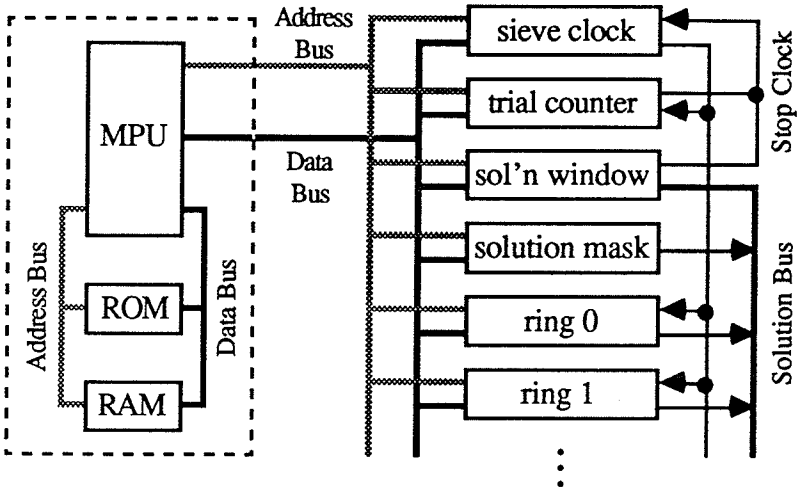


Figure 6-2. Sieve Unit Internal Design

Figure 6-2 shows a simplified outline of a sieve unit's architecture, including its main components and the major signal groups they share. The core of the sieve unit is an 8 bit microprocessor unit (MPU). The MPU carries out the commands transmitted to the sieve unit by its host by controlling the operation of all the other sieve components, either directly or indirectly. The instructions directing the MPU reside in 2K bytes of ROM and make up the sieve's permanent control program, or *firmware*. The microprocessor begins executing its firmware automatically as soon as power is applied to the sieve and continues to execute it for as long as the machine remains powered up. The microprocessor also has access to 128 bytes of RAM which is used as temporary storage by the firmware. The MPU, ROM, and RAM are actually parts of a single chip called a microcomputer unit (MCU).

The MPU reads from and writes to the sieve unit's other components using a conventional bus-oriented approach. The MCU's internal 16 bit address bus and 8 bit data bus are used to access the RAM and ROM, while an external 8 bit address bus and 8 bit data bus access all other components. The latter devices are partitioned into two separate address spaces: a 16 byte space which holds all non-ring components, and a 256 byte address space for the rings. The ring area is further divided into 8 byte segments, each corresponding to one of the 32 ring numbers. Figure 6-3 shows the layout of the two

external address spaces. Not all of the addresses available are used by sieve components; this provides room for new devices should the OAS be upgraded in the future.

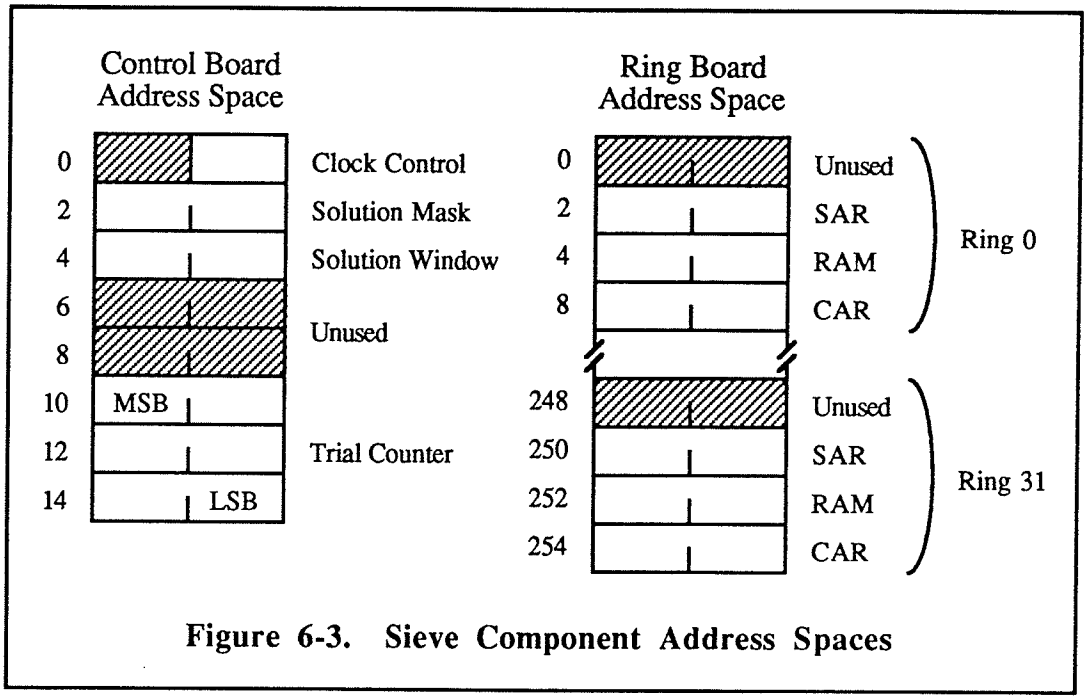


Figure 6-3. Sieve Component Address Spaces

The other major bus used in a sieve unit is a 16 bit *solution bus*. This bus logically ANDs corresponding residues from all rings in the sieve unit, along with the solution mask register bit pattern, producing a "1" on each bus channel that contains a solution. The bus contents can be read by the MPU via the solution window.

All of the major sieve components are tied to a common *sieve clock*, which supplies the stream of high speed clock pulses that regulate sieving. The variable speed clock can be turned on and off by the MPU at will, and is also blocked during sieving the instant a solution occurs or the trial counter overflows. This rapid response allows a sieve unit to sieve at high speed without having to utilize an elaborate pipelined architecture. In addition to its sieving duties, the sieve clock also supplies the clock pulses needed to load the sieve components during idle mode. Not surprisingly, the clock circuit is the most complicated part of the sieve unit design.

The actual arrangement of a sieve unit's hardware is much more complex than Figure 6-2 would indicate, involving many low level details that have been omitted from the diagram for the purpose of clarity. These include the internal design of each sieve component, the serial interface to the host, and the multitude of control signals that govern

the operation of every part of the sieve. Nor can the diagram depict the sequence of events that takes place during its operation. The following sections provide this missing information, and describe the design and operation of the OAS control board, ring board, and backplane, and also that of the sieve unit firmware.

6.2 THE CONTROL BOARD

An OAS control board contains all sieve unit components except the rings. This includes the microcomputer that controls the sieve, the high speed clock used for sieving, the solution detection circuitry, the trial counter, and the host interface.

6.2.1 The Sieve Microcomputer

The control centre for the sieve is a Motorola MC68701 single-chip microcomputer. Containing an MPU, 2K bytes of EPROM, 128 bytes of RAM, and four communication ports within a 40 pin DIP, the MC68701 implements all of the sieve's non-sieving functions in a single, compact package. A complete description of the chip and its capabilities, including many funtions that are not utilized in the OAS sieve unit design, can be found in [Mot84].

The MC68701is automatically initialized whenever power is applied to the OAS. The chip's RESET/V_{pp} pin is held LOW by a 1μF capacitor and remains LOW until the capacitor is fully charged; a 100Ω pull-up resistor then forces the pin HIGH to complete the initialization sequence. Although a sieve unit resets itself properly after a complete power failure, it does not handle a power "glitch" as gracefully. If power fails for less than a few seconds, the capacitor does not have time to fully discharge and the RESET (L) signal is not asserted long enough for the microcomputer to complete its initialization sequence, causing the microcomputer to "hang". A manual initialization switch has been added to the control board to allow the sieve to be reset easily in such an instance.

The microcomputer is brought up in "single chip mode" (mode 7) by three 10KΩ pull-up resistors tied to its mode sense inputs. In this mode, the 8 bit microprocessor begins executing the instructions contained in its 2K on-chip EPROM and continues to execute this code until it is powered off. The microcomputer is clocked by a 4.9152 MHz crystal, giving it a cycle time of approximately 0.8μs. Each instruction takes from two to twelve clock cycles to execute, with the majority requiring no more than four. The MPU uses the

MC68701's communication ports to communicate with the other sieve components and the sieve unit's host by manipulating internal registers under the direction of the firmware.

6.2.2 Host Interface

The sieve unit talks with its host using a standard RS-232C serial communication line connected to the MC68701's built-in serial interface on Port 2. The data format used contains eight bits of data, with one start bit, one stop bit, and no parity. The baud rate used by the interface is derived from the same crystal that drives the MC68701's MPU, which allows the sieve to provide one of four popular rates: 300, 1200, 9600, or 76 800 baud. At the present time, the firmware selects the 9600 baud rate.

A special signal converter chip (Max232) handles the conversion between the RS-232C signal levels used by the serial line and the TTL levels required by the microcomputer's serial interface pins. The capacitors surrounding the Max232 are specially chosen to perform this conversion with only +5V and GND level inputs available; as a result, the sieve does not require the ±12V power levels normally required during RS-232C communications.

6.2.3 Controlling Sieve Components

The operation of the sieve unit hardware is directed by eight *primary control signals* issuing from the MC68701's Port 1 and Port 2. By changing the levels on each of these lines, the chip's firmware can read and write the various sieve components and control the progress of sieving. Since the microcomputer ports can only drive a single TTL load, an eight bit buffer (74F541) is used to increase the fan-out capability of these signals.

Many sieve unit parts do not use the primary control signals directly. Instead, they take in *secondary control signals* formed from one or more of the primary control signals. Most secondary control signals used by control board components are generated by a pair of 20L10 programmable array logic (PAL) chips, which can produce both active LOW and active HIGH signals (the latter by applying De Morgan's Law). A few active HIGH signals that cannot be generated by the PALs directly are generated as active LOW signals and then inverted by a separate chip (74LS04) to achieve the correct polarity.

6.2.3.1 MODE SELECTION

The sieve's mode is specified by a pair of primary control signals (see Table 6.1). The AWAKE (L) line is asserted if the sieve is not in sleeping mode, while I/OMODE (L) is asserted to indicate idle mode. The sieve's firmware puts it into idle mode upon reset.

Table 6.1
Mode Specification

AWAKE (L)	I/OMODE (L)	Mode Selected
LOW	LOW	Idle mode
LOW	HIGH	Problem mode
HIGH	—	Sleeping mode

6.2.3.2 READING AND WRITING COMPONENTS

The micrcomputer reads and writes sieve components using a memory-mapped I/O approach. By placing appropriate values on the sieve's address bus and control lines, the microcomputer transfers data to and from the components over the data bus.

The 8 bit data bus is connected to the MC68701's Port 3, and is amplified using a transceiver (74F545) to improve fan-out. Transferring data from a sieve component to the microcomputer is termed a *read*, while a transfer from the microcomputer to a component is known as a *write*. The direction of a transfer is given by the primary control signal R/$\overline{W}$ ("Read/Write"); a HIGH level indicates a read. All data transfers are done one byte at a time, so multiple I/O operations may be required to access a particular sieve component. Although the sieve hardware ignores the data bus except when an I/O operation is actually in progress, the bus and R/$\overline{W}$ are always configured for reading once the operation has completed.

The 8 bit address bus is connected to the MC68701's Port 4, and is always configured for output; the bus is amplified using a buffer (74F541) to improve fan-out. Whenever an I/O operation is to be done, the microcomputer specifies the address space by asserting the primary control signal RBA (L) ("Ring Board Access") or CBA (L) ("Control Board Access"); when neither line is asserted, the sieve hardware ignores the address bus value. Only the low four bits of the address are decoded for a control board device. Ring board addresses are fully decoded, with the five most significant bits of the address specifying the number of the ring being accessed and the three low order bits specifying the ring part. For all multi-byte components, the lowest numbered byte contains the most significant bits.

Read and write operations are entirely controlled by the microcomputer's firmware and do not make use of the MC68701's ability to access off-chip devices under hardware control. While the latter method would allow components to be read and written in one or two firmware instructions, rather than the sequences of 10 to 25 instructions currently used, the greater freedom that comes with firmware control (as well as the greater ease in correcting design mistakes) was considered more important than speed. Components are normally accessed only at the request of the host, so the limiting factor in performing I/O is the data transmission rate between the sieve and the host rather than the firmware's I/O routines. If necessary, the routines can be optimized to increase their speed.

The firmware uses the following sequence of steps to perform a read operation.

1) Put address of component on address bus.
2) Assert RBA (L) or CBA (L), as appropriate, to select address space.
   (This instructs the component being read to put its data on the data bus.)
3) Read data from data bus.
4) Deassert RBA (L) or CBA (L).

Writing to a component is a bit more complex, since some of them require a clock pulse to load a new value. To generate this pulse, the firmware asserts the primary control signal I/OCLOCK (L) during each write operation. Thus, a write operation is done as follows.

1) Put address of component on address bus.
2) Put data on data bus and configure bus for output.
3) Set R/$\overline{W}$ to LOW to indicate a write operation.
4) Assert RBA (L) or CBA (L), as appropriate, to select address space.
5) Assert I/OCLOCK (L).
   (This forces the sieve's CLOCK line HIGH, loading chips that are clocked.)
6) Deassert I/OCLOCK (L).
7) Deassert RBA (L) or CBA (L).
   (Components which are not clocked latch the data bus when deselected.)
8) Set R/$\overline{W}$ back to HIGH.
9) Configure data bus for input.

Since all sieve components that use the sieve's clock are tied to a single source, each component sees *every* clock pulse that the sieve generates. The secondary control signals for these devices are carefully designed to ensure that at most one device responds to the clock pulse generated during a write operation.

Although the firmware is capable of reading and writing a section of a multi-byte component, it only provides the host with commands to read or write the entire component. It also ensures that I/O operations are only attempted while the sieve is in idle mode, since the components cannot respond in any other mode.

6.2.4 Sieve Clocking

The sieve unit clock circuit supplies the clock pulses needed by sieve components in both idle and problem modes. Each clock cycle begins with the rising edge of the clock signal and continues up to (but not including) the next rising edge. The OAS design allows the clock cycle to be as short as 60 nanoseconds, with a minimum HIGH time of 23.5 ns and a minimum LOW time of 37 ns. The sieve currently runs using a 75 ns clock cycle, of which the first third is spent HIGH.

6.2.4.1 Clock Pulses in Problem Mode

The high speed clock pulses used for sieving originate at the Q output of a J-K flip-flop (74F112). Each clock cycle begins with the flip-flop's "reset" pin asserted, so the clock signal is LOW. A clock pulse generator then deasserts the pin and clocks the flip-flop. If STOPCLOCK (L) is deasserted (ie. the J input is HIGH), the flip-flop is set and the rising edge of the clock pulse is generated; when the clock generator later reasserts the "reset" pin the falling edge is formed (see Figure 6-4 (a)). If STOPCLOCK (L) is asserted when the flip-flop is clocked, the device remains reset and no clock pulse occurs (Figure 6-4 (b)). Thus, the flip-flop generates pulses whose frequency and duty cycle is controlled by the clock generator for as long as STOPCLOCK (L) remains deasserted.

The STOPCLOCK (L) signal is asserted whenever a solution is generated by the sieve's rings or the trial counter reaches its maximum count. The sieve uses an 8 input OR gate (74F30) to combine four GOTSOLN*n* (L) signals from the solution bus and a single PROBDONE (L) signal from the trial counter and then inverts the result using a 2 input NAND gate (74F00) to achieve the correct polarity. This 2 level circuit, along with the circuits that generate the GOTSOLN*n* (L) and PROBDONE (L) signals, operate so quickly that once a stopping condition occurs, STOPCLOCK (L) is asserted before the flip-flop can generate its next clock pulse. On the other hand, the assertion of STOPCLOCK (L) never terminates the current clock pulse prematurely; once the signal's level is sampled at the start of a clock cycle, it is ignored until the flip-flop is clocked at the

start of the next cycle. This design also prevents any "spikes" on the STOPCLOCK (L) line prior to its stabilization from affecting the output of the clock.

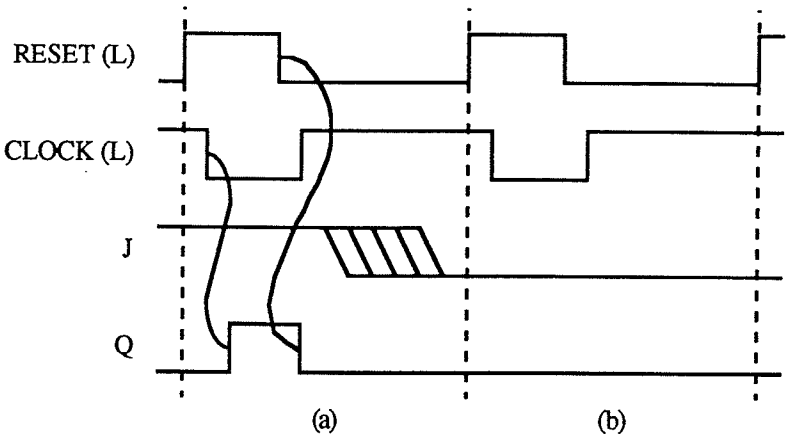


Figure 6-4. Sieve Clock Generation

6.2.4.2 THE CLOCK PULSE GENERATOR

The source of the high speed clock pulses that control the clock circuit's flip-flop is an Advanced Micro Devices clock generator (Am2925). Driven by a 40 MHz oscillator, the chip produces four slower clock signals whose frequency depends on the division factor specified by the three least significant bits of the sieve's clock control register. This factor can be any integer from 3 to 10, providing clock speeds that range from 13.3 MHz down to 4 MHz. Other frequency sets can be obtained by replacing the oscillator, which is socketed for easy removal. The clock circuit's flip-flop uses C3 as its reset input and C4 as its clock input, so that the clock signal used for sieving is HIGH for roughly the same amount of time as C3—from one third to one half of a clock cycle. A complete description of the Am2925 can be found in [Adv85].

The microcomputer starts and stops the Am2925's clock pulses using one of two primary control signals: RUNCLOCK (L) causes the chip to generate a stream of pulses, while STEPCLOCK (L) always generates a single pulse. The sieve's firmware ensures that only one of these signals is asserted at a time and only in problem mode. The Am2925 always completes its current cycle when it is halted, so the firmware can deassert these signals asynchronously without the danger of generating a "runt" pulse. The sieve's clock control register is implemented by an 8 bit latch with read-back capability (74LS793) and

can be read or written by the microcomputer. The register's most significant bit is tied to the Am2925's reset pin, which allows the firmware to initialize the clock circuit during its reset sequence; the other bits of the register are unused.

The Am2925 was originally to have been driven by an oscillator in the 48 to 50 MHz range, providing the OAS with a clock cycle of between 60 and 62.5 nanoseconds during sieving. Although a prototype clock circuit was operated for several months to demonstrate that the chip would run at that speed with a full load of rings, it did not produce a stable signal when later installed on an OAS control board. This unexpected difference is likely due to the fact that the prototype placed the Am2925 under a static load whereas the real rings provide a dynamic load that is much more difficult for the chip to handle at very high speeds. Evidence for this hypothesis was gained by running the sieve using different numbers of ring boards; the maximum speed at which the clock signal was stable turned out to be inversely proportional to the number of boards used.

6.2.4.3 Clock Pulses in Idle Mode

The same clock circuit that produces the high speed clock pulses used in sieving also produces a clock pulse each time the microcomputer writes to a sieve component. The primary control signal I/OCLOCK (L) is tied to the "set" input of the clock circuit's flip-flop. Thus, when the microcomputer asserts I/OCLOCK (L), the flip-flop's Q output is forced HIGH; when the signal is deasserted, the flip-flop output is pulled LOW by the ongoing assertion of its "reset" pin by the clock generator. Any cycle length longer than a few microseconds can be generated by having the firmware assert I/OCLOCK (L) for that interval. Currently, it generates the shortest pulses it can—about 4 μs. Unlike the case with pulses in problem mode, STOPCLOCK (L) has no effect during idle mode. This is an important consideration since the firmware must perform a write operation to remove the conditions that cause STOPCLOCK (L) to be asserted.

6.2.4.4 Clock Signal Distribution

The sieve components that use the clock signal need to receive it with a minimum of delay to ensure that STOPCLOCK (L) is valid by the end of the current clock cycle. Slight differences in the times a clock pulse arrives at the various sieve components ("skewing") are tolerable as long as all chips within a given component receive the pulse simultaneously. To reduce propogation delays and skewing problems, the clock signal is fed through a

single level of amplification rather than a multi-level amplifier tree. A pair of high speed buffer/drivers (74F367) are used to amplify the flip-flop's clock signal output and divide it into five parallel channels: one channel clocks the trial counter, while the other four are fed to the rings. The four ring channels are each capable of driving up to 8 rings. It was originally intended that only a single driver be used, but it provided insufficient internal grounding capability to generate a strong signal; the second chip is mounted directly on top of it to make up the deficiency.

6.2.5 Solution Detection

The OAS solution detection circuitry is outlined in Figure 6-5. Each ring feeds 16 residues into the 16 separate channels of the solution bus, where they are logically ANDed within the bus itself. Each channel is tied to a 110Ω resistor which pulls the line HIGH unless a "0" residue from one or more rings pulls it LOW. The rings produce their residues using open-collector outputs, which can be tied together by the bus without causing electrical problems. The resistor value chosen provides the correct amount of pull-up for a solution line connected to up to 16 rings. This design has two desirable features: not only is it almost instantaneous, but it works with any number of rings. Thus, ring boards can be added or removed from the circuit without requiring changes to the solution detection hardware.

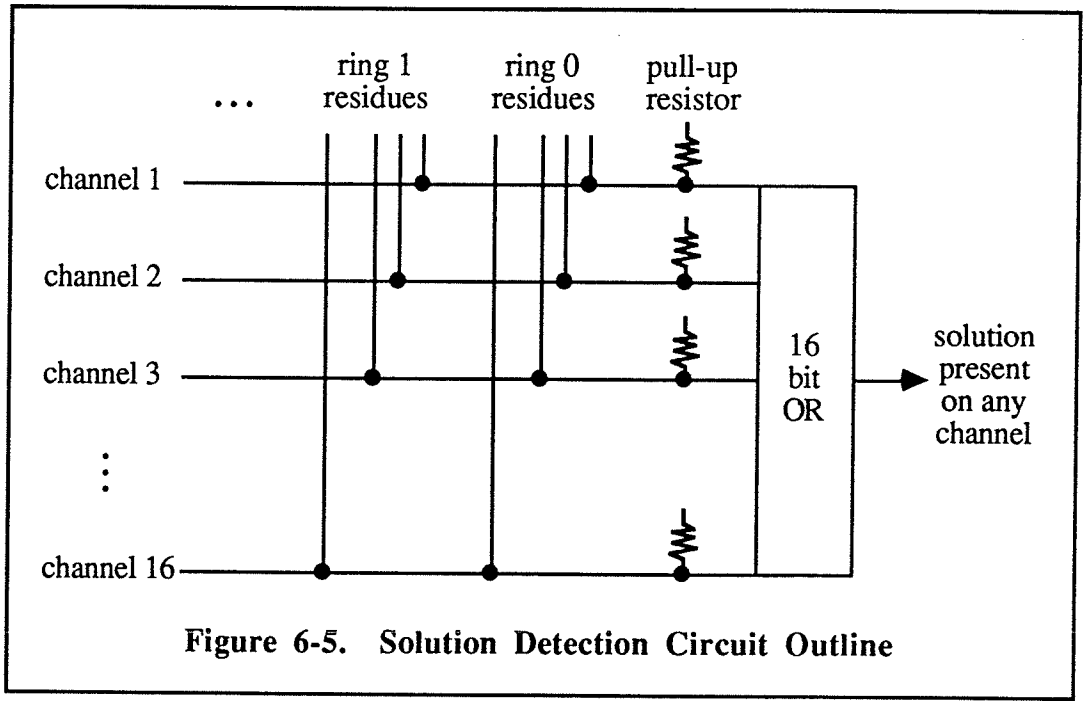


Figure 6-5. Solution Detection Circuit Outline

To permit solutions to halt the sieve clock, the 16 solution bus channels are logically ORed to produce a signal which is HIGH if *any* of them indicate a solution. The 16 bit OR depicted in Figure 6-5 is actually obtained by feeding the channels into a set of four 5 input NOR gates (74F260), producing four GOTSOLN*n* (L) signals, which in turn assert STOPCLOCK (L). The four are also combined by a PAL to generate a SOLUTION (H) signal which is fed into one of the MC68701's Port 1 inputs; this allows the firmware to tell if a solution is present on the solution bus. When a solution is indicated, the microcomputer can determine exactly which bus channel(s) contain a solution by reading the solution window. This activates a pair of 8 bit tri-state buffers (74LS541) that route the solution lines to the data bus.

Not shown in Figure 6-5 is the solution mask that allows any combination of solution bus lines to be disabled by the host. It is implemented as a pair of 8 bit latches with read-back capability (74LS793) which can be both read and written by the microcomputer. The latches' contents are logically ANDed to the solution bus channels by feeding them through a pair of 8 bit latches (74F273) and a pair of 8 bit open-collector buffers (74F621) to the solution bus.

The 74F273 latches provide a means of clearing the solution bus so that sieving can be resumed after a solution has been detected. Writing to the solution window actually clears the latches, which deasserts STOPCLOCK (L) by forcing all solution channels LOW.[1] The host can do this explicitly (the value written to the window is irrelevant), but it is unnecessary: the firmware automatically clears the window whenever the host issues a GO or STEP command. When sieving resumes, the first clock pulse loads the latches with the contents of the solution mask, disabling only the solution lines specified by the host; subsequent clock pulses simply reload the latches. The clock signal used by the 74F273 latches is *not* the same one used by the other sieve components, but a version which does not exhibit I/O pulses during idle mode; this prevents a write operation to another sieve component from "unclearing" the latches after they have been cleared by a write to the window.

---

1 If the trial counter has reached its maximum count STOPCLOCK (L) will remain asserted.

6.2.6 The Trial Counter

The 48 bit trial counter is implemented using six 8 bit counters (74F579). The counter can be read or written by the microcomputer, with each counter segment ignoring clock pulses except when it is actually being written to. Each clock pulse generated while the sieve is in problem mode increments the entire counter as a unit, up to a maximum of $2^{48}-1$ (ie. all "1" bits). Once the counter reaches this value, PROBDONE (L) is asserted to block the sieve clock from generating further pulses.

The trial counter segments are *not* configured as a 48 bit ripple counter, since this would haved incurred a delay of up to 54.5 nanoseconds from the time a clock pulse was received until PROBDONE (L) could be guaranteed valid. Instead, it is implemented as a 40 bit ripple counter concatenated to an 8 bit counter, with the former part incremented if the latter is all "1" bits. Thus, once the high 40 bits become all ones, another 255 clock cycles elapse before the low order counter reaches all ones—plenty of time for the "terminal count" (ie. all "1"s) signal of the 40 bit counter to ripple through its five stages. The terminal count outputs of the 40 bit and 8 bit counters are ANDed by a 74F32 chip to form the final PROBDONE (L) signal, resulting in a delay of at most 15 ns from the time the clock pulse is received.

Since the PROBDONE (L) signal is based on the TC outputs of the counter chips, which are valid only when the chip is enabled for counting, the PROBDONE (L) signal is only valid in problem mode. This means that the microcomputer must temporarily switch the sieve into problem mode (without starting the clock) whenever the firmware wishes to sense the signal during idle mode.

6.3 THE RING BOARD

Each ring board provides its sieve unit with four rings; it also contains a small amount of address decoding logic which ensures that each ring responds only when its assigned number is referenced.

6.3.1 Address Decoding

Each of a board's four rings is enabled by its own RINGSEL*n* (L) signal, which is asserted whenever the microcomputer specifies an address matching one of the ring's

components. The ring combines this signal with the lowest three bits of the address bus, RBA (L), and R/$\overline{W}$ to carry out the desired operation on the ring component being accessed. The "ring select" signals are distributed so that the rings are numbered in sequence with the lowest numbered ring furthest from the backplane.

The RINGSEL*n* (L) signals are generated by a pair of chips at the bottom of the ring board. When the sieve is in idle mode, an 8 bit comparator (74LS521) compares the three most significant bits of the address bus against the three DIP switches set by the user; if they match, it asserts BOARDSEL (L) to indicate that the board contains the ring responding to that address. The "board select" signal is fed into a 2-to-4 bit demultiplexer (74F539) which asserts the appropriate RINGSEL*n* (L) signal based on the next two most significant bits of the address. If BOARDSEL (L) is not asserted, none of the "ring select" signals is enabled. The comparator used in this circuit could easily have been as small as three bits wide; the 8 bit size was utilized simply because chips were readily available.

6.3.2 Ring Design

A shift register sieve, such as UMSU, implements its rings by linking together linear shift registers to form a set of cyclic shift registers of various sizes. To perform sieving, each register is loaded with bits representing the acceptable and unacceptable residue classes for a congruence whose modulus equals its size; a stream of clock pulses is then applied to the chips to circulate the residues past a fixed solution detector. Such rings function quite well, but they do have several drawbacks. First, a large ring uses many chips, which consume a lot of space and power, and which can be difficult to clock simultaneously. Secondly, rings of different sizes must be constructed differently, making design and assembly a very time consuming task. Lastly, a ring can only hold a congruence whose modulus divides the period of the shift register; if no such congruence occurs in a given sieving problem, the ring is useless.

The OAS overcomes these difficulties by implementing the shift register concept in a new way—rather than shifting the residues past a fixed solution tap, it moves the tap past the residues. The main parts of the ring design are outlined in Figure 6-6. Residues are stored in a *t* bit wide *random access memory* (RAM), using "1" bits to indicate acceptable residue classes. Each time a clock pulse is applied to the ring, the *t* residues at the memory location specified by the *current address register* (CAR) are captured by a *residue latch* and placed onto the *solution bus*. The pulse also increments the CAR, causing the residues in

the next RAM location to be captured and tested during the following clock cycle. The CAR does not revert to zero once it reaches its maximum value, M; instead, it loads the value given in the *start address register* (SAR), S. Thus, a sequence of clock pulses causes the ring to cycle through the residues located in the RAM locations from S to M, inclusive, simulating a shift register with $(M-S+1)t$ positions.

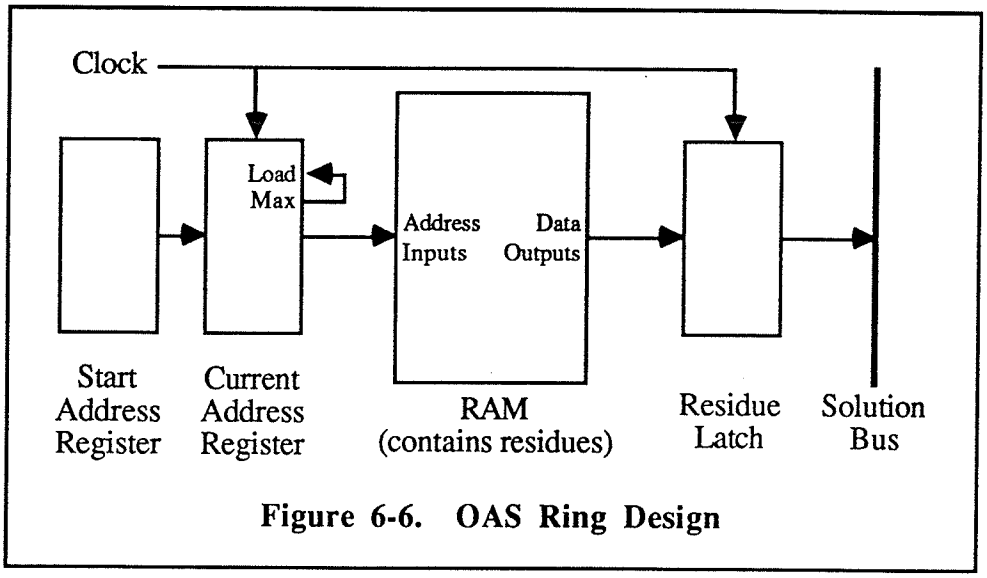


Figure 6-6. OAS Ring Design

The host adjusts the size of the ring to match a given congruence by loading the appropriate starting value into a ring's SAR. Since the congruence's residue classes must occupy an integral number of RAM locations, it is often necessary to load the RAM with multiple copies of the residues to ensure that the total number of residues involved is a multiple of t. For a congruence with modulus m, a total of $\frac{t}{\gcd(m,t)}$ copies are required (see Figure 6-7). As a result, a ring with an n word by t bit RAM can implement any congruence where $\frac{m}{\gcd(m,t)}$ is less than or equal to n; this includes *all* congruences with a modulus not exceeding n and some with a modulus as large as $n \times t$.

The OAS rings use a 16 bit SAR and CAR, giving access to 65 536 RAM locations. The number that can actually be used during sieving depends on the ring's construction. Currently, all rings have an n of 8192 (corresponding to RAM locations $E000_{16}$-$FFFF_{16}$), but this can be incresed to 32 768 (8000_{16}-$FFFF_{16}$) by using larger memory chips. The RAMs and residue latch are also 16 bits wide, providing a t of 16 for each ring.

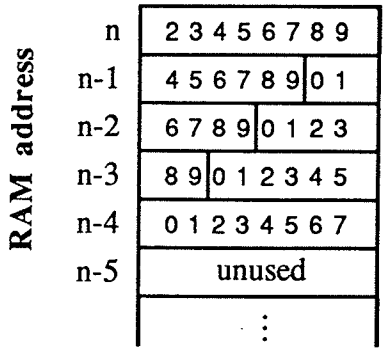


Figure 6-7. Residue Replication in an OAS Ring (m = 10, t = 8)

6.3.3 Ring Implementation

All rings in the OAS are identical in construction and operation. In addition to the four main components discussed below, each ring contains a pair of PAL16L8 chips which convert control signals coming from the control board and from the board's own address decoding logic into the control signals needed by the ring components.

6.3.3.1 START ADDRESS REGISTER

The SAR is composed of a pair of 8 bit latches with read-back capability (74LS793) and is read and written by the microcomputer in the normal manner. The register is not connected to the sieve clock and simply holds its value during problem mode.

6.3.3.2 CURRENT ADDRESS REGISTER

The CAR is a 16 bit ripple counter implemented by a pair of 8 bit counters (74F269). The data inputs of the CAR are tied to the data outputs of the SAR, so the CAR cannot be loaded directly from the data bus. Instead, the microcomputer loads the CAR by loading the SAR with the desired value and then writing to either byte of the CAR; this copies the SAR's contents into the CAR. For convenience, the sieve unit's firmware automatically preserves the existing SAR value whenever the host writes a new value to the CAR. The CAR is read in the normal fashion; a read operation activates a pair of 8 bit tri-state buffers (74LS541) that route the CAR's contents to the data bus.

The CAR ignores all clock pulses during idle mode, except when it is actually being written to. When the sieve unit switches to problem mode, the counters are enabled for counting so that each rising edge of the clock increments the CAR by one. Whenever both stages of the counter reach their terminal count (ie. all "1" bits), the ring's PALs enable the counters for loading, so that the next clock pulse loads the CAR with the SAR value. On subsequent clock pulses, counting continues as before.

6.3.3.3 RING RAMS

A pair of 35ns 8K by 8 bit static RAMs (Cyress 7C185) hold the residues loaded into a ring. Both chips use the outputs of the CAR as their address line inputs, providing a store of 8192 sixteen bit words. Since the RAMs have only 13 address inputs, only the 13 least significant bits of the CAR (A0 through A12) are actually used for addressing; A15 and A14 are left unused, while A13 is tied to an active HIGH chip enable. These three pins must always be set to "1" to ensure the CAR specifies a valid address (ie. any value in the range $E000_{16}$-$FFFF_{16}$). The RAM location addressed by the current CAR value can be read and written by the microcomputer; a pair of 8 bit transceivers (74LS245) provide the necessary interface between the data bus and the RAMs.

The OAS rings are designed so that the existing RAMs can be removed from their sockets and replaced with JEDEC standard 32K by 8 bit RAMs, thereby increasing each ring's capacity by a factor of four. The larger RAMs respond to A13 and A14 (but ignore A15), allowing the CAR to specify addresses in the range 8000_{16}-$FFFF_{16}$. The OAS can support any mixture of 32K and 8K rings without difficulty, so only as many rings as desired need be upgraded. At the time the OAS was built, 32K RAMs were either unavailable, too slow to be used in the sieve, or prohibitively expensive.

When the sieve is switched into problem mode all RAM chips are enabled for reading, causing them to display the contents of the memory location addressed by their associated CAR. There must be a short delay between the time the sieve changes mode and the first high speed clock pulse to allow the RAMs enough time to complete this initial read. Since the RAMs' access time is so much smaller than the microcomputer's instruction cycle time, the firmware provides the necessary separation by simply using two different instructions for changing modes and starting the clock. During problem mode, the residues presented by the RAMs are latched by the rising edge of the sieve clock. This edge also disables the RAMs, since the clock is tied to the chips' active LOW chip enable. On the falling edge of

the clock, the RAMs are reenabled and proceed to access the memory location specified by the CAR. Disabling the RAMs reduces the problem of "soft" errors during read operations by preventing the RAMs from trying to respond while the CAR is changing (see [CSS84]). No matter what speed the clock control register runs the sieve clock at, there is always at least 35 ns between a falling edge of the clock and the next rising edge to give the RAMs sufficient time to produce the correct residues before they are latched. The RAMs are the only sieve components which make use of the falling edge of the sieve clock in problem mode. When the sieve switches back to idle mode, the ring RAMs are disabled and their current residue values are "forgotten" without being captured by the residue latch. The next time the sieve unit goes into problem mode the same residues are re-accessed (unless the host has altered the ring's CAR value) and then latched by the first clock pulse.

The disabling effect of the sieve clock does not interfere with reading and writing of the RAMs during idle mode. Since no clock pulse occurs during a read, the selected RAM remains enabled throughout the operation. During a write, the RAM is actually written to twice: the clock pulse occuring during in mid-operation disables and then reenables the RAM chip. If desired, this redundancy could be eliminated by modifying the sieve unit's firmware to keep it from generating a clock pulse during a write to the ring RAMs.

6.3.3.4 RESIDUE LATCH

The residue latch is composed of a pair of 8 bit latches (74F377) whose outputs are fed into a pair of 8 bit open-collector buffers (74F621). The four chips are functionally equivalent to two 8 bit open-collector latches (MMI 74S383), which are now unavailable. Each latch input is tied to a RAM data line, and each open-collector output is connected to the solution bus. The residue latch is the only one of the four main ring components which cannot be read or written by the microcomputer, either directly or indirectly.

During problem mode, each rising edge of the sieve clock causes the residue latch to capture the 16 residues displayed by the RAMs and place them onto the solution bus for merging with the residues from the other rings. The residue latch holds its current value whenever the sieve clock pulses during idle mode, thus leaving the contents of the solution bus unchanged during I/O operations.

6.3.4 Clocking the Rings

The OAS rings are very sensitive to stray pulses in the sieve clock signal. A single extraneous pulse is sufficient to put a ring out of synchronization with the rest of the sieve and make further sieving meaningless. A major cause of noise in a high speed signal like the clock is small reflections that occur whenever the signal is split in two. Although the individual reflections that occur at various points on the OAS ring boards are negligible, the regular placement of clock signal traces combines the reflections as they move toward the backplane; eventually the growing echo triggers the ring counters that it passes. Thus, if the OAS is run using unmodified boards, its rings became desynchronized almost immediately, with the amount of error proportional to the ring's closeness to the backplane.

The first attempt to cure this problem involved trying to eliminate the reflections entirely by tying a terminating resistor of the appropriate size at each point where the clock signal split; unfortunately, the ring boards' lack of a ground plane made it difficult to determine the clock signal's impedance and thus calculate the correct resistor size, so this approach had to be abandoned. Instead, the main clock traces were cut and replaced by lengths of wire-wrap wire which transmit the clock signal from the backplane to each ring in parallel. This arrangement reduces the number of times the clock signal is split, and thus the number of reflections. As an extra precaution, the clock signal going to each ring is passed through a 68Ω resistor, which helps absorb any reflection coming back down the wire and prevents it from affecting the other rings.

Although modifying the clock signal paths reduced the reflection problem significantly, the sieve never operated reliably until the problem of "ground bounce" was solved (see §6.1.1, "OAS Construction"). Increasing the ring boards' grounding capacity improved the behavior of the rings' counters so much that it suggests that the reflections caused by the layout of the clock traces are no longer a significant problem, and that the extra wires, capacitors, and resistors added to the ring boards are now unnecessary. Not surprisingly, no attempt was made to test this hypothesis by removing these modifications.

6.4 THE BACKPLANE

All of the boards in a sieve unit are connected to the backplane by a pair of 108 pin connectors designed for use with the sieve's card cage. The backplane provides a channel for each signal that must pass between the control board and its ring boards. These are:

1) I/OMODE (L), which specifies the sieve's mode,
2) the I/O control signals: R/$\overline{W}$ and RBA (L),
3) the 8 bit address bus,
4) the 8 bit data bus,
5) the 16 bit solution bus,
6) the sieve clock signal, which is transmitted on four identical channels.

All of these signals originate on the control board and flow into the ring boards, except for the the data bus, which is bidirectional, and the solution bus, whose signals flow from the rings into the control board. The backplane provides special shielding for the clock signals and the solution bus to limit the "ringing" caused by their high switching rates during sieving. The other lines do not require such protection since they only change state during idle mode and are controlled by the comparatively slow sieve unit firmware.

Since the OAS can support multiple sieve units, each acting independently, the channels provided by the backplane are not permanently connected to every board in the card cage. Instead, each channel is broken up into sections which connect two adjacent boards, allowing boards to be "daisy chained". Every board in the sieve has the freedom to utilize any signal sent to it from a neighbouring slot and to propogate it to the opposite neighbour. By designing both the control board and the ring board to sample and propogate signals correctly, the boards within the card cage can be made to form sieve units of varying sizes, which can also be temporarily merged into larger units when needed.

6.4.1 Control Board Interface

A control board handles the signals travelling through the backplane in two different ways. In both idle mode and problem mode, the board communicates with its ring boards by driving or sensing the backplane channels that are "downstream" from it (ie. in the adjacent higher numbered slots) and ignores any signals coming from the adjacent sieve unit "upstream". In sleeping mode, the board does the opposite: it pays no attention to the values on the backplane channels and simply propogates any incoming signal in the appropriate direction.

For some backplane signals, this dual capability is implemented by routing the control board source of the signal and the upstream source through separate tri-state devices, only one of which is actived at a time. The data bus, the address bus, and the control signals I/OMODE (L), R/$\overline{W}$, and RBA (L) are all directed to the rings in this manner, using a pair

of tranceivers (74F545), a pair of buffers (74F541), and a dual 4 bit buffer (74F241), respectively. These devices also serve to strengthen their signals, either amplifying the output of the MC68701 communication ports (which can only drive a single TTL load) or acting as a repeater for signals originating in a control board further up the backplane.

The sieve clock and solution bus signals are not fed through tri-state devices, since this would introduce delays that would degrade the sieve's performance, particularly if multiple delays were introduced by merging several sieve units. Instead, each signal is fed through a relay (Potter & Brumfield JWD-171-25) that reflects the sieve's mode. If the control board is active, the relays are opened to prevent contact with any control board upstream; if the board is sleeping, they are closed to provide a delay-free bridge between the upstream control board and the rings downstream. Since no amplification takes place, the clock signal generated by the active control board in a merged sieve unit has been designed with enough capacity to drive all of the rings under its control without aid. To completely isolate a sleeping control board from the backplane, the board also turns off its clock signal amplifier (74F367) to prevent it from interfering with the clock signal coming from the active control board, and disables the open-collector outputs of its solution mask register (74F621s) to prevent the register from masking out solutions that the active control board wishes to see.

6.4.2 Ring Board Interface

Because a ring board operates in the same way whether or not it is part of a merged sieve unit, its interface to the backplane is much simpler than for a control board. Each ring board has permanent connections to the backplane channels coming from the control board and simply propogates each to the next ring board or control board (with one exception).

The I/OMODE (L), R/$\overline{W}$, and RBA (L) signals are taken from the backplane and amplified by a buffer (74F541) before being passed on to the rings and address decoding circuitry. Each address bus line is sent to exactly one chip, which may be the buffer, the address decoding comparator (74LS521), or the decoding demultiplexor (74F539). The data bus is connected to an amplifying transceiver (74F545) which is enabled only when I/O is actually being performed on one of the board's rings. To avoid introducing delays, the clock signal is taken from one of the four backplane channels by means of a jumper and distributed to all four rings without amplification. For the same reason, the open-collector

residue outputs of each ring are tied together and fed into the backplane's solution bus channels without amplification.

With the exception of the RBA (L) signal, each signal received by the board is propogated without amplification; RBA is fed through the address decoding demultiplexor, meaning that an assertion of the signal is propogated only if the ring board decides not to respond to the I/O request itself. Passing the signal on in this instance would not have caused a problem for the ring boards further down the backplane, since they would not try to respond to the request, but a sleeping control board that saw the signal would activate its data bus transceiver to permit a data transfer to or from the rings downstream from it; if a read operation was being performed, both the selected ring board and the sleeping control board would try to drive the data bus at the same time, which would damage the chips involved. By having the ring board propogate only I/O requests it cannot respond to itself, a sleeping control board is prevented from reacting to requests that its rings cannot possibly service. As a bonus, this arrangement means that only one ring board can ever respond to an I/O request, even if several boards have the same settings on their DIP switches; this ensures that the sieve hardware is not damaged by having two or more rings driving the data bus simultaneously.

6.5 FIRMWARE OVERVIEW

The OAS firmware is a simple command interpreter and monitor, designed along the lines of Motorola's MIKBUG or ASSIST09 programs. However, rather than providing an environment for developing application programs for the microcomputer on which it runs, the OAS firmware provides an interface to a sieve unit's sieving hardware that allows its host to solve a sieving problem by issuing a series of sieving commands. The firmware executes each command by manipulating the appropriate parts of the sieve unit's hardware and returns any results that are generated.

The source code for the OAS firmware is written in Motorola 6801 assembly language and is approximately 2100 lines long, including comments. When assembled, its object code occupies 2045 of the MC68701's 2048 bytes of EPROM. Forty-seven of the chip's 128 bytes of RAM are used for permanent variables that maintain their values until changed; the remaining RAM locations form a run-time stack which is used by various routines to preserve information about their caller's environment and for holding the values of local variables. The fact that the microcomputer's EPROM can be easily erased and

reprogrammed makes it easy to correct or upgrade a sieve unit's command set, and this was done repeatedly during the development of OASiS. The current version of the OAS firmware is called "SIVMON 3.8".

The relatively small amount of memory provided by the microcomputer greatly limits the functions the firmware is able to implement. As a result, the program concentrates on essential services. For example, the host is not permitted to backspace over and correct typing errors since the OASiS software should never make such mistakes. The few frills that do exist were usually added to benefit a human host who has to debug the sieve unit hardware, and these were implemented only when space was available.[2] Although the program was coded with an eye towards compactness in order to squeeze as much functionality as possible into a small amount of memory, the primary consideration during design was always clarity. Consequently, the program exhibits a well structured, modular design that is easy to understand and to modify.

6.5.1 Program Structure

The code that makes up the OAS firmware can be divided into five types of routines: a mainline, command and subcommand bodies, SIVMON services, routines that control the sieve hardware, and routines that handle low level host communications.

The mainline is invoked automatically whenever the MC68701 is reset. It first initializes the microcomputer's interfaces to the host and the sieve components, sets the sieve's default mode and clock speed, and issues a "wake up" message to the host. Once initialization is complete, the mainline enters an infinite loop which prompts for a command character and then invokes the appropriate command body to interpret it; when the command is complete, the mainline prompts for another command character.

A command body can be short or long, depending on the command being implemented. Simple commands are carried out by manipulating one or more of the program's global variables, calling a subroutine to manipulate the sieve hardware, requesting a service that reads a numeric argument from the host or writes a result to it, or any combination of these actions. More complex commands usually require the command body to request a service

---

2 A number of these frills later turned out to be useful capabilities that are exploited by the OASiS software. For example, the ability to suspend command output and to abort a command in mid-execution were both found to be useful in handling ring verification errors.

that gets an additional command character from the host and invokes the appropriate subcommand body. The subcommand body performs its own set of actions, which may include invoking a further level of subcommand, and so on. Although subcommand nesting can be done to as many levels as necessary, SIVMON never goes beyond sub-subcommands.

The routines that control the sieve hardware and those that implement the various SIVMON services can be invoked by the mainline or a (sub)command body. The former are simple subroutines that manipulate the sieve hardware by using the MC68701's parallel communication ports to generate the necessary control signals. The SIVMON services implement 16 commonly required command invocation and high level host communication functions, and are essentially subroutines that are invoked using a special calling mechanism. Both the service mechanism and a number of the host communication services provided are modelled after the services of Motorola's ASSIST09 program [Mot81]. The subroutines that implement SIVMON's low level host communication protocol are usually invoked as part of a request for one of the host communication services, but the mainline also calls the routines directly in a couple of instances to perform non-standard operations.

The processing of commands by the mainline and the various command bodies is driven by input from the host. Each input character is processed as it is needed; if a routine needs to get a character from the host and none is available, it waits for one to arrive. A similar pause also occurs if the program wants to send a character to the host but the host is not ready to receive it. If the sieve hardware is sieving during such a pause, and solution counting is enabled, SIVMON spends the idle time checking the solution window for solutions; each time one appears, the program increments a solution counter kept in RAM and restarts sieving. Thus, the firmware can count solutions in the background even as it executes commands from the host in the normal manner.

6.5.2 Command Processing

The mainline prepares for the execution of a command by prompting the host with a '>' character. If the sieve is in sleeping mode or idle mode, the mainline takes the next input character and invokes the appropriate command body. However, if the sieve is in problem mode, the mainline spends the time between the end of the prompt and the arrival of the command character watching the sieve hardware; if the sieve stops—either because a solution has occurred (if in recording mode) or the trial counter has reached its maximum

count (in either search mode)—it informs the host by transmitting a '!' character and then continues waiting for input without doing the additional checking. The checking is also abandoned if the command character arrives first; thus, a command is never interrupted once the firmware begins interpreting it.

The mainline begins the actual execution of a command by requesting the DOCMD service and passing it the address of a *command table*. The service routine gets an input character from the host, searches the table for the matching character, and invokes the corresponding command body. If the input character does not match any entry in the command table, DOCMD sends an error message to the host and returns to the mainline. The mainline prevents the host from issuing a command that is not permitted in the current sieve mode by having a different command table for each mode and passing DOCMD the appropriate table address.

Many command bodies utilize the host communication services to read in numeric argument values or to write out results as they carry out the command. When a command body finishes its work, it calls the ENDCMD service to return to its caller (ie. the mainline). If it needs to invoke a subcommand, a command body requests the DOSCMD service and supplies a command table; the service then operates exactly as the DOCMD service does for the mainline. A subcommand body is essentially the same as a command body and also ends with a call to ENDCMD. This returns control to the subcommand body's caller, permitting a command body to invoke several subcommands in a single command if it wishes (this is never done in the current version of SIVMON).

The only way for a (sub)command body to encounter an error condition is as the result of bad host input. To simplify and standardize the handling of errors, SIVMON's services automatically abort a command (rather than returning a failure indicator to the caller and letting it handle the situation) if:

1) a DOCMD or DOSCMD request does not find an entry for the current input character in the specified (sub)command table,
2) a hexadecimal input request encounters an invalid digit,
3) a CANCEL character is encountered during any input or output request.

The error handler restores the microcomputer's stack to the level it was at when the command was invoked, issues an appropriate error message to the host, and returns control to the mainline. For errors that the services cannot detect, such as a numeric argument that is too large or too small, the (sub)command body does its own error checking. In such

cases, the body issues an error message and returns to the mainline in the normal manner via the ENDCMD service.

6.5.3 SIVMON Services

The various SIVMON service routines are implemented as software interrupt handlers rather than subroutines. This mechanism is used solely to save space in the MC68701's limited EPROM, and relieves each service routine of the need to preserve the caller's environment by using the microcomputer's interrupt servicing capability to automatically save the caller's registers on the stack when a request is made and restore them when the service is complete. This saves six to ten bytes in each of the sixteen service routines, at the cost of having a single 31 byte interrupt handler. The interrupt mechanism also saves a byte each time a service is requested by allowing a request to be made using two bytes rather than the three required for a subroutine call, thus saving an additional 82 bytes throughout the entire program.

Each service is requested by executing a "Software Interrupt" (SWI) instruction, which stacks the microcomputer's registers and jumps to a common service dispatcher; the dispatcher invokes the appropriate service routine body based on the value of the the byte following the SWI instruction. When the service has been carried out, the routine returns control to the caller by executing a "Return From Interrupt" (RTI) instruction, which restores all of the microcomputer's registers from the stack. A caller passes arguments to a service routine using one or more of the available registers and receives its results in the same way; unless the routine deliberately overwrites one or more of the register values saved on the stack during the course of the service, the caller's registers are left unchanged by the request.

The sixteen services provided by SIVMON are listed in Table 6.2. The first four services control the invocation and termination of SIVMON commands and subcommands, and have been described in the preceding section. The next eleven services deal with host communication and allow the caller to read and write single characters, character strings, and numeric values in hexadecimal form. The final service causes SIVMON to examine the sieve hardware and increment its RAM solution counter if necessary.

Table 6.2
SIVMON Services

Name	Code	Function
INTCMD	0	Initialize command handler
DOCMD	1	Invoke command using command table
DOSCMD	2	Invoke subcommand using subcommand table
ENDCMD	3	Return from command or subcommand body
GETCHR	4	Get a character from the host
GETH	5	Get a 1 digit hexadecimal input value
GET2H	6	Get a 2 digit hexadecimal input value
GET4H	7	Get a 4 digit hexadecimal input value
PUTCHR	8	Send an output character to the host
PUTSP	9	Send a space character
PUT2H	10	Send a byte as 2 hexadecimal characters
PUT4H	11	Send a word as 4 hexadecimal characters
PDATA1	12	Send a NUL-terminated string of characters
PDATA	13	Do a PCRLF, then a PDATA1
PCRLF	14	Send carriage return and linefeed characters
DOBACK	15	Do background solution counting

6.5.4 Host Communication

SIVMON implements a simple, full-duplex communication protocol that also gives the host XON/XOFF control over output from the sieve.

Each character transmitted by the host arrives asynchronously and is stored in an eight character buffer until it is needed by a command routine. The arrival of an input character activates an interrupt-driven input handler, which immediately copies the character to the buffer to prevent it from being lost if the host transmits another character before the first one is used by the firmware. If the buffer overflows, all of the characters it contains are lost; however, SIVMON's command routines normally consume input characters as fast as the host can generate them and the buffer rarely (if ever) has more than one character in it at any time. Characters are taken from the buffer on a FIFO basis by one of the four input services. If the buffer is empty at the time an input character is needed, the input routine waits for one to appear before continuing. Each character is echoed back to the host as it is removed from the buffer.

Three input characters have special meanings to the OAS firmware and are not handled in the normal manner—the characters CANCEL (ASCII code 3), XOFF (code 19), and XON (code 17). If the host sends a CANCEL character, the firmware does not place it in the input buffer; instead, it discards any unprocessed characters lying in the buffer and sets

a flag variable. When the command processor next requests an input or output service, the service routine sees that the flag is set and aborts the current command. Similarly, the firmware records the arrival of an XOFF or XON character by setting another flag variable to indicate if output has been suspended by the host. This flag is examined only by the I/O service routines whenever a character is to be transmitted to the host.

The output services write to the host without the use of interrupts or a buffer and simply transmit characters to the host as they are generated by SIVMON. If a character cannot be transmitted, either because the microcomputer's serial port is busy transmitting the previous one or because the host has blocked output by sending an XOFF character, the output routine waits for the blocking condition to disappear and then continues.

Chapter 7 :
Some OASiS Results

Dripping water hollows out a stone,
A ring is worn away by use.

Ovid, *IV.x.5*

The Open Architecture Sieve System became operational in January, 1989. Since then, it has been put to work on a variety of problems in number theory, including pseudo-powers, periodic continued fractions with long periods, and quadratic polynomials with a high density of prime values. This chapter describes these problems and the results obtained.

7.1 Problems Involving Quadratic Character

Let $p_1, p_2, \ldots, p_m$ be a set of odd primes and $\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_m$ be a sequence with $\varepsilon_i^2 = 1$. We are interested in integers whose quadratic character with respect to each p_i is specified by the corresponding ε_i; that is, we wish to find N such that

$$(N/p_i) = \varepsilon_i, \quad i = 1, 2, \ldots, m,$$

where (a/b) is the Legendre symbol. This is a natural sieving problem in which each equation is represented by a congruence that specifies either the quadratic residues or non-residues modulo p_i for $\varepsilon_i = 1$ and -1, respectively. Several instances of this problem are of special interest because they have applications to other branches of number theory. Previous work in this area includes work by Lehmer, Lehmer, and Shanks [LLS70] and Shanks [Sha73].

In the problems examined below, p_i is the i^{th} odd prime (ie. $p_1 = 3, p_2 = 5, \ldots$) and all ε_i are equal. Following Shanks, aR_p (aN_p) denotes the set of positive $N \neq n^2$ of the form $8k + a$ which are non-zero quadratic residues (non-residues) of all odd primes $q \leq p$. Similarly, $-aR_p$ and $-aN_p$ denote the sets for which $-N$ has the specified quadratic character, rather than N itself. In each problem, OASiS searched for N values belonging to one or more of these classes. Because the congruences modulo 3 and 8 contain only a single acceptable residue, OASiS was able to increase its hardware sieving rate by a factor of 24 (to approximately 5.12×10^9 trials per second) by combining them into one congruence and optimizing its search.

7.1.1 Pseudo-squares

The class $1R_p$ represents non-square integers, N, where $N \equiv 1 \pmod 8$ and $(N/q) = 1$ for all primes $q \leq p$. Since these values appear to have the quadratic character of a perfect square for primes up to p, but are not perfect squares, they are known as *pseudo-squares*. Pseudo-squares can be used in a variety of applications, including a test for primality [Hal33] and an inexpensive test for a perfect square [Cob66].

We denote the smallest member of the class $1R_p$ as N_p. The N_p values from N_2 to N_{127} were found by Kraitchik [Kra24], Lehmer [Leh28], [Leh54], and Lehmer, Lehmer, and Shanks [LLS70], while unpublished calculations by Lehmer and others by Williams have found N_{131} through N_{191}. OASiS has now been used to find N_{193} through N_{223}. A complete list of N_p values and their sources is given in Table 7-1.

OASiS searched for members of the class $1R_{157}$ from 10^{14} to 1.25×10^{16}; these were later tested to determine if each value was also an element of $1R_p$ for larger values of p. The class $1R_{157}$ was selected since it loads the sieve hardware with as many congruences as possible, and it provided a reasonable balance between generating too few solutions to be of interest and generating too many for the sieve's host minicomputer to process easily.

Since many odd perfect squares are also solutions to the problem's congruences, a simple filter program was used to eliminate them as they were generated by the sieve. Although the work involved in testing each solution candidate was not great, the numerous squares encountered meant that up to 80% of the host's CPU time was spent filtering, which greatly slowed the progress of sieving. The sieve usually found its next solution candidate long before the filter program had finished testing the previous one, forcing the machine to spend up to 92% of its time idle. As well, the low priority level given to the OASiS software (designed to prevent it from degrading performance for the host's other users) reduced the effective sieving rate even further during peak periods. As the search progressed, the decreasing density of squares improved the filtering situation considerably, but by the time it had reached 10^{16} OASiS was still only running at 65% of its full speed. In all, the interval from 10^{14} to 1.25×10^{16} required 1933 hours to search, for an average sieving rate of 1.78×10^9 trials per second (35% of its theoretical limit). Had the OASiS software been extended to take advantage of the sieve's firmware solution counting capability, it might have been possible to use the technique described by Lehmer, Lehmer, and Shanks [LLS70] to eliminate the filter program entirely and achieve the maximum sieving rate much earlier.

Table 7-1.
Least Pseudo-squares

# of primes	p	N_p	Source
1	2	17	Kraitchik (1924)
2	3	73	moveable strips
3	5	241	
4	7	1 009	
5	11	2 641	
6	13	8 089	
7	17	18 001	
8	19	53 881	
9	23	87 481	
10	29	117 049	
11	31	515 761	
12	37	1 083 289	
13	41	3 206 641	
14	43	3 818 929	
15	47	9 257 329	
16	53	22 000 801	Lehmer (1928)
17	59	48 473 881	bicycle chains
18	61	"	
19	67	175 244 281	Lehmer (1954)
20	71	427 733 329	SWAC
21	73	"	
22	79	898 716 289	
23	83	2 805 544 681	Lehmer, Lehmer,
24	89	"	Shanks (1970)
25	97	"	DLS-127
26	101	10 310 263 441	
27	103	23 616 331 489	
28	107	85 157 610 409	
29	109	"	
30	113	196 265 095 009	
31	127	"	
32	131	2 871 842 842 801	Lehmer (1973)
33	137	"	DLS-157
34	139	"	[unpublished]
35	149	26 250 887 023 729	
36	151	"	
37	157	112 434 732 901 969	Williams (1988)
38	163	"	UMSU
39	167	"	[unpublished]
40	173	178 936 222 537 081	
41	179	"	
42	181	696 161 110 209 049	
43	191	"	
44	193	2 854 909 648 103 881	Stephens (1989)
45	197	6 450 045 516 630 769	OASiS
46	199	"	
47	211	11 641 399 247 947 921	
48	223	"	

We can make two observations about the entries in Table 7-1. First, Bach [Bac89] has shown under the generalized Reimann Hypothesis (GRH) that if G is a proper multiplicative subgroup of the integers modulo m, then there exists some $n > 0$ such that $n \notin G$ and $n < 2(\log m)^2$. Thus, if N_p is a prime and r is the least prime such that $(N_p/r) = -1$, then we should have $r < 2(\log N_p)^2$, or

$$N_p > e^{\sqrt{r/2}}.$$

In fact, all of the entries in Table 7-1 greatly exceed this lower bound. Second, the numbers in the first part of the table only work for a single p, while later values tend to work for two or more consecutive primes. Since there is as yet no explanation for this distribution, it will be interesting to see if this tendency continues for values beyond N_{223}.

7.1.2 Negative Pseudo-squares

The class $-7R_p$ represents integers, N, where $-N \equiv 1 \pmod 8$ and $(-N/q) = 1$ for all primes $q \le p$. Since the negatives of the N values have the same quadratic character as the values in the previous section, they can be thought of as *negative pseudo-squares*.

As before, the smallest member of the class is denoted by N_p. A table of N_p values for $p \le 131$ is given by Lehmer, Lehmer, and Shanks in [LLS70]. Shanks [Sha73] later determined the values up to N_{163}, but only one value was published. OASiS has now been used to extend the table up to N_{211}. A complete list of the N_p values is given in Table 7-2.

OASiS searched for members of the class $-7R_{157}$ from 0 to 5×10^{15}; the values generated were then tested to determine N_p for $157 \le p \le 211$. A second, smaller search generated the entries from the class $-7R_{137}$ between 0 and 5×10^{13}, filling in N_{137} through N_{151}. Since 7 is not a quadratic residue modulo 8, perfect squares were not generated by these searches; thus, the filter program used in finding positive pseudo-squares was not required and OASiS was able to sieve at full speed throughout. Consequently, the first search took about 12 days to perform and the second about three hours.

Negative pseudo-squares are used in the primality test of Selfridge and Weinberger (see §21 of [Wil78]). The result of Bach asserts that a value N need only be tested against a relatively small number of primes (fewer than $2(\log N)^2$ in all) to ensure it is a prime, making this method an effective, polynomial-time primality test under the GRH. The fact that all of the values in Table 7-2 exceed Bach's bound provides evidence in support of this result that does not depend on the truth of the GRH.

Table 7-2.
Least Negative Pseudo-squares

# of primes	p	N_p	Source
1	2	7	Lehmer, Lehmer,
2	3	23	Shanks (1970)
3	5	71	DLS-127
4	7	311	
5	11	479	
6	13	1 559	
7	17	5 711	
8	19	10 559	
9	23	18 191	
10	29	31 391	
11	31	307 271	
12	37	366 791	
13	41	"	
14	43	2 155 919	
15	47	"	
16	53	"	
17	59	6 077 111	
18	61	"	
19	67	98 538 359	
20	71	120 293 879	
21	73	131 486 759	
22	79	"	
23	83	508 095 719	
24	89	2 570 169 839	
25	97	"	
26	101	"	
27	103	"	
28	107	"	
29	109	"	
30	113	328 878 692 999	
31	127	"	
32	131	513 928 659 191	
33	137	844 276 851 239	Stephens (1989)
34	139	1 043 702 750 999	OASiS
35	149	4 306 732 833 311	
36	151	8 402 847 753 431	
37	157	47 375 970 146 951	
38	163	52 717 232 543 951	
39	167	100 535 431 791 791	
40	173	251 109 340 045 079	
41	179	493 092 541 684 679	
42	181	"	
43	191	"	
44	193	1 088 144 332 169 831	
45	197	"	
46	199	"	
47	211	"	
48	223	> 5 000 000 000 000 000	

7.1.3 Periodic Continued Fractions with Long Periods

Williams [Wil81] has pointed out that if D is square-free, then the period length of the continued fraction expansion of $\sqrt{D}$, $p(D)$, should be bounded above by an expression of the form $c\sqrt{D}\log\log D$. In particular, if

$$f(D) = \begin{cases} \sqrt{D}\log\log D & \text{for } D \equiv 1 \pmod 8, \\ \sqrt{D}\log\log 4D & \text{otherwise,} \end{cases}$$

then it should be true that

$$G(D) = p(D)/f(D) < k + o(1)$$

under the extended Riemann Hypothesis for ζ_K when $K = Q(\sqrt{D})$. Here $k = 3.7012$, but Lévy's Law indicates that it could be as small as $12e^{\gamma}\log 2/\pi^2 \approx 1.50103$. Patterson and Williams [PW85] examined many large $G(D)$ values to see how they approached this bound. The largest value they found occurred for $D = 13\ 518\ 648\ 471\ 574$ when $G(D)$ was equal to 1.081381.

Earlier work by Williams [Wil81] suggests that D values for which $G(D)$ is large are most likely integers of the form q or $2q$, where q is a prime and $q \equiv -1 \pmod 4$. It is also desirable to have $(D/r_i) = 1$ for the odd primes $r_1, r_2, \ldots, r_n$ where n is as large as possible. Thus, candidate D values can be found by solving a system of linear congruences.

Following the example set in [PW85], OASiS was used to find D values of four types:

(i) $D \equiv 3 \pmod 8$ D prime,
(ii) $D \equiv 7 \pmod 8$ D prime,
(iii) $D \equiv 6 \pmod 8$ $D/2$ prime,
(iv) $D \equiv 1 \pmod 8$ D prime,

the type (iv) D values being examined to determine whether their associated G-values start to catch up with the larger values obtained by the other three classes, as predicted by Shanks. For the first three cases, OASiS searched the range 0 to 5×10^{15} for members of the classes $3R_{157}$, $7R_{157}$, and $6R_{157}$, respectively; each search took about 12 days. For the type (iv) case, we simply reused the results from the earlier pseudo-square search, examining members of the class $1R_{157}$ within the range from 10^{14} to 10^{16}.

Calculating the exact value of $G(D)$ for all of the D values generated by OASiS would have required many hours of computer time due to the repeated need to determine $p(D)$ using the continued fraction algorithm which has a time complexity of $O(D^{1/2+\varepsilon})$. Instead, the $O(D^{1/4+\varepsilon})$ "large step" algorithm utilized by Stephens and Williams [SW88] was used to calculate the regulator of $Q(\sqrt{D})$, $R(D)$, from which

$$G'(D) = R(D)/f(D)$$

was determined. Since $R(D)$ is expected by Lévy's Law to be proportional to $p(D)$, a large $G'(D)$ value is a likely indicator of a large $G(D)$. Using this as a guide, we then calculated $p(D)$ and $G(D)$ for a few of the D values generated. Each regulator calculation averaged less than a second of computer time on an Amdahl 5870 computer.

Tables 7-3 through 7-6 list a subset of the D values generated for each of the problems run by OASiS, along with their corresponding p- and G-values. Each D is shown only if $G(D)$ exceeds the value of $G(d)$ for all computed values of d of the same type with $d < D$. These results extend the work of [PW85] by about an order of magnitude, and give no indication that the projected limit of $k \approx 1.50103$ will be exceeded. The growth of $G(D)$ for type (iv) D values also appears to be slightly greater than for the other three types, indicating that it is indeed catching up to them.

7.1.4 Quadratic Polynomials Generating Many Primes

Let $f_A(x) = x^2 + x + A$ $(A \in \mathbf{Z}, A > 0)$ and let $P_A(n)$ be the number of prime values assumed by $f_A(x)$ for $x = 0, 1, 2, \ldots, n$. Polynomials for which $P_A(n)$ is relatively large have interested mathematicians since the time of Euler, who found that $P_{41}(39) = 40$. Since that time, other examples have come to light which generate an even higher proportion of primes as n approaches infinity.

The search for polynomials of this type is aided by considering Hardy and Littlewood's conjecture F [HL23] which asserts that

$$(*) \qquad P_A(n) \sim C(D)L_A(n)$$

where

$$D = 1 - 4A,\ L_A(n) = 2\int_0^n \frac{dx}{\log f_A(x)},\ \text{and}\ C(D) = \prod_{p \geq 3}\left(1 - \frac{(D/p)}{p-1}\right),$$

the latter product being taken over all odd primes p. If the conjecture holds, then $f_A(x)$ has a high asymptotic density of prime values whenever $C(D)$ is large. Considerable evidence in support of this has been given by Shanks [Sha59], [Sha60], [Sha63] and Fung and Williams [FW89].

Table 7-3.
***D* - Type (i)**

D	*R(D)*	*p(D)*	*G(D)*
25 969 254 121 099	11 325 477.6	9 551 418	0.539479
35 474 768 258 491	26 025 216.6	21 942 306	1.057447
121 521 362 355 331	48 737 550.2	41 084 814	1.058503
152 290 419 440 611	54 891 577.2	46 274 886	1.062983
206 546 921 647 291	64 474 167.2	54 350 198	1.069334
820 362 746 906 299	131 207 980.2	110 604 710	1.079902
924 401 140 322 059	140 451 569.1	118 399 282	1.087996
4 976 709 946 053 091	330 581 613.0	278 655 786	1.089614

Table 7-4.
***D* - Type (ii)**

D	*R(D)*	*p(D)*	*G(D)*
963 864 514 519	2 107 959.8	1 778 716	0.538151
46 257 585 588 439	30 459 726.7	25 679 652	1.081244
289 358 196 053 551	77 740 401.4	65 539 148	1.086442
1 135 360 188 709 399	156 220 850.7	131 687 940	1.090171

Table 7-5.
***D* - Type (iii)**

D	*R(D)*	*p(D)*	*G(D)*
27 266 351 212 006	1 448 975.4	1 219 624	0.067199
42 940 991 222 614	14 450 735.1	12 182 504	0.532732
48 888 369 417 694	30 359 479.5	25 594 764	1.047768
129 143 129 979 406	50 279 000.9	42 391 356	1.058905
193 289 509 403 566	62 162 428.3	52 405 772	1.066434
256 397 742 215 806	71 809 801.1	60 536 004	1.067108
285 278 695 393 246	76 061 142.8	64 119 584	1.070606
477 747 574 223 494	99 581 194.9	83 943 714	1.078595
600 206 879 107 606	112 267 896.3	94 642 190	1.082969
4 518 102 473 256 934	313 418 266.5	264 191 526	1.084990
4 826 678 427 841 846	326 631 901.1	275 319 106	1.093416

Table 7-6.
***D* - Type (iv)**

D	*R(D)*	*p(D)*	*G(D)*
112 434 732 901 969	45 498 660.0	38 364 413	1.040660
162 516 480 029 401	54 895 119.1	46 286 149	1.040928
273 323 976 657 169	71 812 592.8	60 545 353	1.045206
457 165 855 430 761	94 198 806.8	79 417 945	1.055462
570 395 076 767 569	105 696 894.2	89 110 088	1.058265
732 376 785 497 449	120 433 435.9	101 538 843	1.061985
3 135 969 368 926 969	252 416 665.9	212 781 805	1.062959
4 113 071 509 075 489	290 678 761.5	245 039 927	1.066602

Assuming (*), the task of locating an A for which $C(D)$ is likely to be large is not hard. For $f_A(x)$ to assume prime values, A must be odd; thus, $-D = 4A - 1 \equiv 3 \pmod 8$. To maximize $C(D)$, it is necessary to have $(D/p) = -1$ for as many small primes p as possible. As pointed out by Lehmer [Leh36], this restriction ensures p does not divide $f_A(x)$ for any x; so if it applies for many primes then $f_A(x)$ is likely to be a prime. Hence, if N is a member of the class $-3N_p$ then $D = -N$ is likely to have a large $C(D)$ value.

A set of candidate N values was generated by having OASiS find the members of the class $-3N_{173}$ within the range 0 to 5×10^{15}, a task which took about 12 days. Since the infinite product method of calculating $C(D)$ converges quite slowly, Fung and Williams' modified "giant step-baby step" method was used to evaluate $C(D)$ for each D. This approach is very fast but is contingent on the extended Riemann Hypothesis.

Table 7-7 lists a subset of the D values generated by OASiS, along with their corresponding $C(D)$ and $q(D)$ values. The latter value denotes the least prime such that $(D/q(D)) \neq -1$. Only D values greater than 10^{15} for which $q(D) \geq 191$ are shown. These results can be compared with a similar table given in [FW89] which extends up to 10^{15}.

The data in the table can be used to locate N_p, the smallest member of the class $-3N_p$. The values of N_2 through N_{197} are known to be less than 10^{15} and are listed in [FW89]. A quick glance at Table 7-7 shows that $N_{199} = 4\ 311\ 527\ 414\ 591\ 923$ and that the next missing entry, N_{211}, must exceed 5×10^{15}.

Table 7-7 also illustrates that a large $q(D)$ value does not necessarily guarantee a large $C(D)$ value. In fact, $-N_{199}$ has one of the lowest $C(D)$ values found, indicating that it is a non-residue for relatively few primes beyond $p = 199$ when compared with the other table entries. Nor is the converse true: of the ten D values found by OASiS whose $C(D)$ exceeds 4.9 (see Table 7-8), only two appear in Table 7-7 and a full half of them are members of $-3N_{173}$ but not of the more restrictive classes from $-3N_{179}$ on. Their large $C(D)$ values indicate that they have a relatively high proportion of non-residues for primes beyond 173.

The D value $-2\ 068\ 660\ 612\ 674\ 307$ is particularly noteworthy because its $C(D)$ value of 5.0978921 is the highest ever found, surpassing the record of 5.0894316 discovered by Fung and Williams for $D = -531\ 497\ 118\ 115\ 723$. There may be other values $< 5 \times 10^{15}$ which have a larger $C(D)$, but they will be members of classes more general than $-3N_{173}$ and correspondingly difficult to isolate. For the present, if Conjecture F holds,

$$x^2 + x + 517165153168577$$

can be said to have the highest known asymptotic density of primes of any polynomial of that form. Additional support for the truth of the conjecture comes from the fact that

$$\frac{P_{517165153168577}(10^6)}{L_{517165153168577}(10^6)} = \frac{300923}{59031.829} = 5.0976398,$$

which is quite close to $C(-2068660612674307)$, as predicted.

Table 7-7.
***D*-values where $q(D) \geq 191$**

D	*C*(*D*)	*q*(*D*)
–1 936 187 977 599 283	4.6499685	191
–2 075 334 218 440 387	4.4565543	191
–2 080 538 415 662 947	4.8777963	191
–2 280 563 801 157 163	4.8505941	191
–2 443 638 146 871 667	4.5308361	191
–2 572 394 636 095 243	4.5732258	191
–2 621 536 114 644 283	4.5566321	191
–2 667 258 747 189 523	4.5473160	191
–2 692 385 915 777 083	4.7797430	193
–2 773 901 630 544 907	4.5808188	191
–2 980 855 020 126 547	4.4013186	191
–3 126 717 241 727 227	4.5685162	193
–3 127 258 244 646 187	4.6359081	193
–3 149 000 695 013 947	4.7666727	191
–3 306 963 252 448 003	4.5539217	193
–3 349 571 159 430 787	4.8028039	193
–3 361 682 073 539 827	4.7758725	191
–3 426 547 415 712 283	4.6784043	191
–3 501 931 983 774 547	4.6213901	191
–3 567 911 491 464 643	4.4899719	193
–3 728 839 196 991 283	4.5471659	193
–3 794 312 952 243 643	4.6797966	191
–3 820 909 447 274 203	4.6906107	193
–4 087 860 124 363 723	4.9285914	191
–4 234 760 613 525 187	5.0181916	197
–4 311 527 414 591 923	4.5293043	211
–4 499 600 282 582 827	4.6037822	197
–4 692 944 772 737 323	4.7691045	193
–4 700 459 864 595 763	4.6313214	191
–4 867 291 923 996 643	4.6462439	193

Table 7-8.
***D*-values with $C(D) \geq 4.9$**

D	$C(D)$	$q(D)$
–2 185 525 654 774 603	4.9186918	179
–1 841 263 497 634 507	4.9220531	181
–4 087 860 124 363 723	4.9285914	191
–2 685 474 212 955 307	4.9375363	179
–2 206 536 270 988 603	4.9451552	179
–1 557 482 259 009 307	4.9774707	179
–3 411 659 915 168 563	5.0031973	179
–4 234 760 613 525 187	5.0181916	197
–4 342 220 938 996 627	5.0302734	181
–2 068 660 612 674 307	5.0978921	181

7.2 PSEUDO-CUBES

Another sieving problem, and one that is particularly well suited to the capabilities of OASiS, is the search for *pseudo-cubes*. These are analogous to the pseudo-squares seen earlier, and are defined to be those integers of the form $9k \pm 1$ which are (non-zero) cubic residues of all primes less than or equal to some prime p, but are not perfect cubes. Although a natural sieving problem, it has received far less attention than pseudo-squares; the only previous work is an unpublished table of least pseudo-cubes up to N_{139} generated by Cobham using a conventional computer search [Cob68]. OASiS has now been used to duplicate and extend this work up to N_{313}. The results are summarized in Table 7-9. Since every number is a cubic residue modulo p if $p \equiv 2 \pmod 3$, N_p always equals the preceding table entry, N_q, unless $p \mid N_q$, so to save space the table omits entries for any $p \equiv 2 \pmod 3$ unless N_p differs from N_q. The only instance of this phenomenon for $p \leq 313$ occurs when $N_{83} \neq N_{79}$; it apparently went unnoticed by Cobham, and the correct value for N_{83} appears here for the first time.

OASiS searched for N_p values by running a series of problems, each of which began at the point where the previous one left off and included an additional congruence. Since the congruences with $p \equiv 1 \pmod 3$ exclude about two-thirds of their residue classes, while the congruences with $p \equiv 2 \pmod 3$ exclude only one, only the former were actually loaded into the sieve; the remainder were simulated in software by the host. This greatly increased the time between solutions, and both increased the average sieving rate and reduced the number of solution candidates tested by the host. It is worth noting that a sieve with a fixed set of ring sizes would not have been able to perform this optimization. For example, UMSU would only have loaded 14 "desirable" congruences into hardware, whereas

OASiS was able to load up to 28. OASiS also doubled its normal sieving rate to 4.25×10^8 trials per second by taking advantage of the single residue congruence modulo 2 to optimize its search. Since some perfect cubes are also solutions to the congruences used in the problems, a simple filter program was used to reject them. However, unlike the pseudo-square case, relatively few perfect cubes were encountered, and they did not significantly affect the progress of sieving. Hence, OASiS was able to search the interval from 0 to 10^{15} in about 700 hours. An example of a pseudo-cube problem file, the results obtained, and the filter routine used is given in Appendix G, "A Sample Problem".

Table 7-9.
Least Pseudo-cubes

# of primes	p	N_p	Source
* 1	2	17	Cobham (1968)
4	7	71	computer search
6	13	181	
8	19	2 393	
11	31	3 457	
12	37	5 669	
14	43	74 339	
18	61	"	
19	67	166 249	
21	73	2 275 181	
22	79	7 235 857	
* 23	83	7 298 927	
25	97	8 721 539	
27	103	"	
29	109	91 246 121	
31	127	"	
34	139	98 018 803	
36	151	1 612 383 137	Stephens (1989)
37	157	"	OASiS
38	163	7 991 083 927	
42	181	"	
44	193	"	
46	199	20 365 764 119	
47	211	2 515 598 768 717	
48	223	6 440 555 721 601	
50	229	29 135 874 901 141	
53	241	"	
58	271	"	
59	277	406 540 676 672 677	
61	283	"	
63	307	"	
65	313	"	
67	331	$\geq$ 1 006 698 672 458 725	

All primes except those marked with * are $\equiv 1 \pmod 3$

Chapter 8 : Beyond the OASiS

All things began in order, so shall they end,
and so shall they begin again

Sir Thomas Browne

This chapter analyzes the performance of the Open Architecture Sieve System and looks at areas where further development is possible, including multi-processor sieving. It also describes a new sieving system being developed at the University of Calgary.

8.1 AN OASiS RETROSPECTIVE

The development of the Open Architecture Sieve System was an attempt to produce a sieving system that would set new standards in speed and flexibility. Now that the system is operational, it is possible to re-examine its six main design and performance goals to see how well they have been satisfied.

1) *Generality.* OASiS can solve almost any sieving problem that the user cares to define, with few limits on the form of the problem's congruences and none at all on any extra restrictions that it imposes.
2) *Speed.* The OAS runs at a rate at nearly 215 million trials per second, making it the fastest sieve yet built. When OASiS is given a problem containing congruences with only a single acceptable residue class, this rate can be boosted to billions of trials per second.
3) *Rings.* The sieve provides 16 programmable size rings, each capable of holding a congruence whose modulus can be as large as 8192. The rings have already been used to run problems that would not have been possible on a sieve with fixed ring sizes. The sieve's ability to implement very large congruences is also inviting, but has not yet been used other than as a means of placing more than one congruence into a ring.
4) *Reliability.* In the eleven months that OASiS has been in operation (as of September 1989), the host software has demonstrated its ability to run problems for long periods without user intervention and its ability to tolerate error conditions, including system crashes. The OAS itself has been very reliable: to date there has never been a recorded instance of hardware error during sieving.

5) *Portability*. The system's hardware and software are designed for portability. Exactly how portable they really are won't be known until OASiS is moved to a new host, but some modifications have already been made to the host software without difficulty.
6) *Flexibility*. The design of the OAS allows future hardware modifications to be made easily, including the addition of extra processors and more or larger rings.

Thus, as far as can be tested, the OASiS project appears to have achieved all of its goals, and in some cases has even surpassed them.

Despite these achievements, a few aspects of the system's design have not turned out as well as first envisioned. The only real failure is the OAS' inability to meet the original target sieving speed of 256 million trials per second, which, as mentioned previously, was primarily due to our inexperience in designing high speed hardware and budget limitations; the rest of the system has generally performed as intended. One area that did cause some problems was the serial communication interface between the sieve and its host. Chosen to provide good portability, its limited bandwidth restricts the flow of data in both directions so that the time taken to load or verify a typical sieving problem borders on the unreasonable (about five minutes for each operation) and problems that transmit many solution candidates to the host run noticeably slower than those that transmit relatively few. At present, the existing interface is satisfactory, but it is a potential weakness that may hamper an expanded version of OASiS that needs to perform more I/O. Future sieve designers are advised to pay as much attention to getting data in and out of the sieve as they do to manipulating the data once it has arrived at its destination.

In the final analysis, the success of any project is best judged by two criteria: does it perform the job it was designed to do and does it do so at a reasonable cost? OASiS scores well on both counts, solving sieving problems with unmatched speed and flexibility at the price of a small workstation. If OASiS is used continuously over a five year period, the cost of its development and operation will work out to about $5 per day—a bargain when compared against the costs of using a mainframe or supercomputer to solve the same problems over the same period of time.

8.2 FUTURE DEVELOPMENT OF OASIS

Although OASiS is now up and running, its development is by no means complete. Areas needing further examination include methods of using the existing sieve hardware

more effectively and the introduction of an expanded version of the OAS utilizing multiple sieve processors.

8.2.1 Improving the Existing System

Much of the effort expended during the development of OASiS was driven by the need to get a functional system in place as quickly as possible. Now that some experience has been gained by running real problems, attention may be turned to those areas of the system that do not function as well as they could or which were omitted entirely.

8.2.1.1 Improving Sieving Optimization

Wherever possible, the OASiS software increases the effective sieving rate of the OAS hardware by merging all congruences having a single acceptable residue and then having the sieve test only those values that lie in the single acceptable residue class of the resulting congruence. The software simply loads the remaining congruences into the rings so that adjacent residue positions in a ring differ by the modulus of the merged congruence rather than by one. Extending this technique to examine only the acceptable residue classes of a congruence having n acceptable residues out of m would allow OASiS to increase its sieving performance by a factor of $\frac{m}{n}$. A typical problem involving quadratic residues or non-residues would run over five times faster by merging the two congruences having two or three acceptable residues into a single congruence with six acceptable residue classes and programming the sieve to test these classes only.

Modifying the OASiS software to perform this optimization requires it to select the congruences to be optimized, merge them, and load the remaining congruences into the sieve so that adjacent residue positions reflect the gaps created by omitting the unacceptable residue classes. Reordering the residues of an arbitrary congruence (mod p) produces a congruence (mod lcm(n,p)) that contains $\frac{n}{\gcd(n,p)}$ copies of each residue. Figure 8-1 illustrates the effects of this process on an arbitrary congruence (mod 5) when a congruence with $m = 7$ and $n = 2$ is used as the basis of the optimization. As before, the ring corresponding to the congruence being used as the basis of the optimization contains only acceptable residues, but the residues in the other ring now repeat with a period that is twice as long as before. The larger congruences that are generated by the optimization means that

the technique cannot be applied to a sieve with fixed size rings unless n divides the size of the ring for all rings in the device,[1] so a sieve such as UMSU could not handle even the simple case involving only two residue classes. Since the sizes of the OAS rings can be set by the host, the only effect of the larger congruences on OASiS would be a reduction in the number of congruences that could be loaded into the sieve. In some cases, the larger congruences formed through optimization would occupy no more ring space than the unoptimized congruence since the ring's internal design would require the presence of multiple copies of its residues anyway.

		x - 0	1	2	3	4	5	6	7	8	9	10 ...
(a)	$x \equiv ?$ (mod 5)	mod 5 - $?_0$	$?_1$	$?_2$	$?_3$	$?_4$	$?_0$	$?_1$	$?_2$	$?_3$	$?_4$	$?_0$ ...
	$x \equiv 2,5$ (mod 7)	mod 7 - $\times_0$	$\times_1$	$\surd_2$	$\times_3$	$\times_4$	$\surd_5$	$\times_6$	$\times_0$	$\times_1$	$\surd_2$	$\times_3$ ...

standard ordering of residues

		x - 2	5	9	12	16	19	23	26	30	33	37 ...
(b)	$x \equiv ?$ (mod 5)	mod 5 - $?_2$	$?_0$	$?_4$	$?_2$	$?_1$	$?_4$	$?_3$	$?_1$	$?_0$	$?_3$	$?_2$ ...
	$x \equiv 2,5$ (mod 7)	mod 7 - $\surd_2$	$\surd_5$	$\surd_2$	$\surd_5$	$\surd_2$	$\surd_5$	$\surd_2$	$\surd_5$	$\surd_2$	$\surd_5$	$\surd_2$ ...

optimized ordering of residues

Figure 8-1. Two-residue Optimization

8.2.1.2 Improving Ring Utilization

The OASiS software does not always fill the available rings with congruences as well as it might and currently insists that all rings have the same maximum size. Modifying the software to use an algorithm that produces a closer approximation to the optimal fitting would increase the system's ability to handle large problems or allow it to run a problem using fewer rings than are currently required. Removing the "same size rings" restriction is of less importance, but is likely to be important if the OAS rings are upgraded in the future. A good starting point for finding a better congruence mapping algorithm would be to examine algorithms that solve the closely related problem of scheduling tasks on a multi-processor computer system.

---

1 Alternatively, the host software could choose to leave a ring empty if the larger congruence will not fit.

Even with a better mapping algorithm, there will always be sieving problems where the arrangment chosen by OASiS is unacceptable because of peculiarities specific to the problem. While the existing problem file syntax allows the user to override the default fitting, manually specifying where each congruence should go is tedious work. A better approach would be to allow the user to assign a priority level to each congruence which would guide the OASiS software into deriving an acceptable mapping by itself.

8.2.1.3 Implementing Solution Counting

The OASiS software does not currently take advantage of the solution counting mode provided by the OAS. This omission is seldom important since few sieving problems require solution counting, but its presence would be a definite asset for those that do. Before beginning to implement this facility, it would be necessary to decide whether to forbid counting for a problem that requires the host to implement congruences in software rings or perform filtering, since OASiS would not be able to take advantage of the sieve's solution counting mode and would sieve no faster than if it were recording the problem's solutions.

8.2.1.4 Improving Host/Sieve Communication

As mentioned, the limited bandwidth of the serial link between the sieve and its host presents difficulties when a problem requires a great deal of interaction between the two machines. There are several ways in which this difficulty could be alleviated, some of which involve only relatively simple changes to the existing system.

The most obvious solution is to increase the baud rate of the serial link to 19.2K or beyond, but this is not easily done. The sieve's serial interface is closely tied to the microcomputer's internal clock, so increasing the baud rate might require the use of a crystal that would actually decrease the rate at which commands are processed. This problem could be overcome by altering the control board design to allow the MC68701 to use an external clock source, but it would not be a trivial modification.

Even with a faster baud rate, a number of communication problems would remain. Currently, the host waits for the sieve to echo each character before sending the next; thus, it sits idle for a significant amount of time when transmitting a command. For commands involving a lot of characters, as with the loading or verification of rings, the effective rate

of transmission is closer to 1200 baud than 9600. Altering the sieve's firmware and host software to use a half-duplex protocol would allow the host to send characters in bursts without waiting for acknowledgement from the sieve and would decrease the amount of idle time it experienced. The sieve's microcomputer could probably keep up with the resulting flood of data in most situations, but XON/XOFF control would have to be introduced to prevent data from being lost when it couldn't.

If necessary, a further increase in the communication rate could be achieved by sending (possibly unprintable) characters containing eight residues apiece during the loading and verification of rings, rather than the current system of printable characters containing four residues. This would halve the number of characters transmitted during these commands, but would also limit the ability of the user to transmit commands to the sieve hardware from an ordinary terminal, hindering debugging. Even worse, it would limit portability because not all computers support 8 bit serial communications. Rewriting the sieve's firmware to handle only 6 or 7 residues per character would be much more complicated than getting it to handle eight residues, and might result in the loading and verification of rings being no faster than before.

As a last resort, the OAS could be modified to use an 8 bit parallel interface instead of a serial interface. This decision would involve making major changes to the OAS hardware and firmware, and lesser changes to the OASiS software, but it would also allow communication to be performed as fast as the two machines could generate and process data. The portability problems that this type of interface generates are well known from the experience of the UMSU system.

8.2.2 Multi-processor Sieving

The modular design of the Open Architecture Sieve opens the door to the construction of a sieving system with multiple sieve processors. By using n processors, the OAS could take advantage of the inherently parallel nature of sieving to increase the rate of sieving by a factor of n.

8.2.2.1 Parallel Processing Strategies

The simplest method of achieving parallelism would be to partition a sieving problem into n subproblems, each of which searches one n^{th} of the original search range. The

problems would be run on the n processors, after which their results would be combined to obtain the results for the original problem. Getting OASiS to support such a capability would entail adding a command to partition the work and submit the resulting subproblems for execution. The host's problem scheduling mechanism would also have to be modified to run a sieving problem on each of the available sieve units simultaneously. If desired, a command could also be added to handle the collation of the results of the subproblems, rather than requiring the user to do these operations manually. One bonus of this approach to parallelism is that the new scheduler would also allow totally unrelated problems to be solved simultaneously on separate sieve units.

If the end of the search range is not known, as is often the case in sieving problems, the interval partitioning method of achieving parallelism is unusable. Instead, OASiS would have to direct the processors to search the interval in an interleaved fashion, causing the n processors to act like a single sieve with $16n$ solution taps: each would test 16 values out of $16n$ consecutive possibilities and leave the remainder to the other n–1 processors. Modifying the existing SIEVE command to control all n processors would be a non-trivial task, and special care would be required to synchronize the operation of the sieves since they would still be physically independent devices.

Regardless of the form of parallelism adopted, its introduction raises the problem of determining how to utilize the sieve's rings effectively. At the moment, ring allocation is trivial: each problem run uses all of the rings. However, once multiple processors become available, the user will be forced to tailor the arrangment of processors and rings to maximize the number of processors without flodding the host with solution candidates. If rings of different maximum sizes are available, the resource allocation problem is even more difficult to solve. Unless some means can be found to measure the hardware requirements of a typical sieving problem, it is impossible to know how many sieve processors or rings to construct in the multi-processor version of the OAS.

8.2.2.2 SIMULATING MULTIPLE SIEVE UNITS

The unique design of the OAS rings can be utilized to allow a single sieve unit to work on two or more related problems simultaneously. Although a ring is normally loaded with the residues for a single congruence, it can also be partitioned into "logical rings", each containing a different set of residues for the congruence. Figure 8-2 illustrates an example in which six taps of a ring test one set of residues (modulo 12), while the remaining ten

taps test a different set. If analgous configurations are established in all rings, a single sieve unit can solve two distinct problems at the same time, one at $\frac{6}{16}$ of the normal rate and the other at $\frac{10}{16}$. In general, up to 16 different problems could be solved by the OAS; the only requirement would be that the problems share the same set of congruence moduli.

Whether such an exotic approach to parallel processing would be of any practical use is unclear. There are certainly problems that require solving sets of congruences with common moduli but distinct sets of residues, such as the search for pseudo-n^{th} powers, but they could probably be handled more easily by simply running a number of separate jobs in the normal way. A more realistic use for this capability would be to divide each ring in half and run the same problem twice; although this would cut the sieving rate in half, the host could be programmed to check that the two halves produced identical results, greatly reducing the chances of a transient hardware error going undetected.

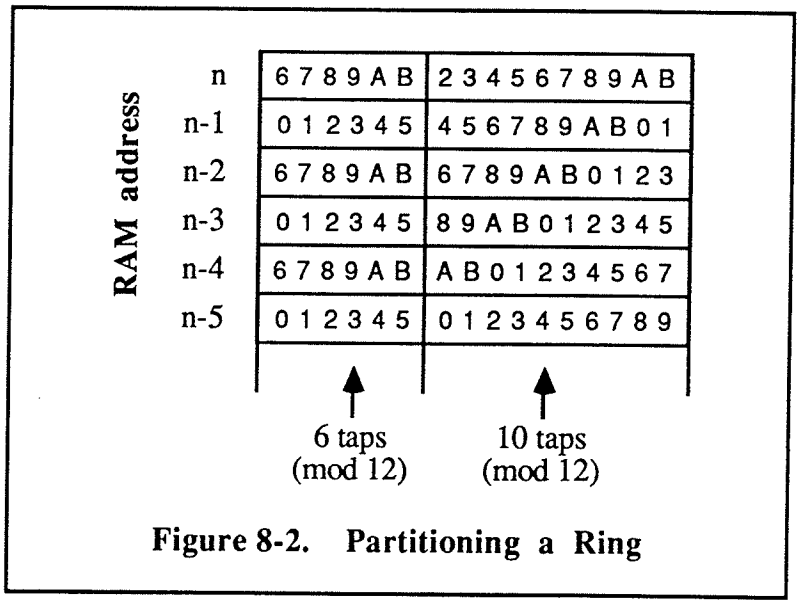


Figure 8-2. Partitioning a Ring

8.2.3 A "Second Generation" Sieve Unit

Although a multi-processor sieve could be constructed by simply duplicating the current batch of OAS circuit boards, the number of ring boards needed to implement a sieve unit would allow no more than two or three processors to be installed before the sieve's card cage was filled. Instead, it would probably be worthwhile to increase the OAS' multi-processing capabilities by building a set of "second generation" sieve units that are approximately 50% smaller than the original model. At the same time, several flaws in the

original boards could be corrected to make the new units about 20% faster than the old one. The redesign process would not alter the overall design and operation of the sieve unit, so large sections of the existing boards and all of the OAS firmware could be reused "as is".

The original 16 MHz specification for the sieve clock could be reached by making a couple of significant changes to the OAS boards. First, the clock circuit based on the Am2925 chip would have to be replaced by one composed of FAST TTL components to permit a 16 MHz signal to be generated reliably. This would involve either building a stripped down version of the Am2925 that provides only the functions actually used in the sieve or designing a totally new clock circuit; in either case, one of the current generation of programmable array logic (PAL) devices could be used as the basis for the circuit. Second, the sieve's circuit boards would have to be redesigned with additional layers for power and ground planes, taking special precautions in placing the clock line traces to avoid the signal reflections encountered previously. The circuit board design tools used to build the first version of the OAS are capable of supporting such four layer boards.

The size of a sieve unit could be decreased by redesigning the rings. The current 16 chip ring design could be reduced to just three chips by combining all of the ring's non-RAM functions into a single chip containing the SAR, CAR, and solution latch components, as well as generating the ring's control signals and providing the interface to the data and solution buses. The new ring chip could probably be implemented using existing forms of application specific integrated circuit (ASIC)—either a semi-custom programmable array or a more expensive custom VLSI chip. While it would be nice to reduce the ring design to a single chip by also integrating the RAM into the chip, attempting to match the very high capacities provided by existing memory chips would not be cost effective. The much smaller footprint of the new ring design would permit the construction of a shorter ring board holding at least eight rings, rather than the four rings found on existing boards. This design would also lessen the distance the clock signal travels to reach the most distant ring on a ring board so that the distance no longer exceeds safe design limits for a 16 MHz signal. Because the new rings would be functionally identical to the old ones, they would be compatible with the existing sieve hardware and firmware.

8.3 THE UNIVERSITY OF CALGARY SIEVE

In 1988, two graduate students at the University of Calgary, Michael Hermann and Cameron Patterson, announced a proposal to develop a high performance sieving device

based on custom VLSI hardware [HP88]. Further details appear in [HP89] and describe a system that would increase the current record for sieving speed by an order of magnitude.

The design of the new sieve combines features of both the UMSU system and OASiS. Like the OAS, the new sieve is to be controlled by a microcomputer and use a serial link to its host to increase its portability. The powerful Motorola 68000 processor chosen as its controller is intended to allow it to implement in software any congruences that are not present in hardware, without relying on the host. As with UMSU, the sieve is to have fixed size rings (whose moduli represent the first 30 primes) and use eight taps per ring; combined with a 25 MHz clock frequency, the initial version of the sieve will sieve 200 million values per second. By using custom VLSI technology, all of the sieve's rings and solution detection circuitry are to be contained on a single chip; the sieve itself will consist of a single printed circuit board with room for up to 16 such chips. A fully configured sieve with all of its sockets filled would thus provide 128 solution taps per ring and be capable of sieving an unprecedented 3.2 *billion* values per second in hardware. Its designers hope to have the device in operation sometime in 1990.

References

Making a book is a craft, as is making a clock;
it takes more than wit to become an author.

Jean de la Bruye're

[Adv85] Advanced Micro Devices, Inc. *Bipolar Microprocessor Logic and Interface Data Book.* Sunnyvale: Advanced Micro Devices, Inc., 1985.

[Bac89] Bach, E. "What To Do Until the Witness Comes: Explicit Bounds for Primality Testing and Related Problems." *Mathematics of Computation* (to appear).

[Ber85] Bernoulli, Joh. *Joh. Heinrich Lamberts...deutscher gelehrter Briefwechsel.* Vol. 5. Berlin: n.p., 1785.

[BLS75] Brillhart, John, and D. H. Lehmer, J. L. Selfridge. "New Primality Criteria and Factorizations of $2^m \pm 1$." *Mathematics of Computation* 29 (1975): 620-647.

[BLSTW88] Brillhart, John, and D. H. Lehmer, J. L. Selfridge, Bryant Tuckerman, S. S. Wagstaff, Jr. *Factorizations of $b^n \pm 1$, $b = 2, 3, 5, 6, 7, 10, 11, 12$ up to high powers.* Vol 22 of *Contemporary Mathematics.* 2nd ed. Providence: American Mathematical Society, 1988.

[Bri81] Brillhart, John. "Fermat's Factoring Method and Its Variants." *Congressus Numerantium* 32 (1981): 29-48.

[BS67] Brillhart, John, and J. L. Selfridge. "Some Factorizations of $2^n \pm 1$ and Related Results." *Mathematics of Computation* 21 (1967): 87-96.

[Car33] "Machine Performs Difficult Mathematical Calculations." *Carnegie Institution of Washington - News Service Bulletin* 3, no. 3 (March 12, 1933): 19-22.

[CEFT62] Cantor, D. G., and G. Estrin, A. S. Fraenkel, R. Turn. "A Very High-Speed Digital Number Sieve." *Mathematics of Computation* 16 (1962): 141-154.

[Cob66] Cobham, Alan. "The Recognition Problem for the Set of Perfect Squares." IBM Research Paper, R.C. 1704. 26 April 1966.

[Cob68] Cobham, Alan. Letter to Drs. D. H. and E. Lehmer. 4 March 1968.

[CSS84] Chappell, Schuster, Sai-Halasz. "Stability and Soft Error Rates of SRAM Cells." *ISSCC Digest of Technical Papers* (February 1984): 162-163.

[Dic19] Dickson, L. E. *History of the Theory of Numbers.* Vol . 1. 1919. Reprint. New York: Chelsea Publishing, 1966.

[Dig86] Digital Equipment Corporation. *VAX/VMS User's Manual*. Maynard: Digital Equipment Corportation, 1986.

[Don74] Donzelli, Mario. "Theory and Design of a High Speed Electronic Sieve." M.Sc. diss., University of Illinois, 1974.

[FW89] Fung, G. W., and H. C. Williams. "Quadratic Polynomials Which Have a High Density of Prime Values." To appear.

[GC66] Gauss, C. F. *Disquisitiones Arithmeticæ*. Translated by Arthur A. Clarke, S.J. New Haven: Yale University Press, 1966.

[GJ79] Garey, Michael R., and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco: W. H. Freeman, 1979.

[Gla78] Glaisher, J. W. L. "On factor tables, with an account of the mode of formation of the factor table for the fourth million." *Proceedings of the Cambridge Philosophical Society* 3 (1878): 99-138.

[Hal33] Hall, Marshall. "Quadratic Residues in Factorization." *Bulletin of the American Mathematical Society* 39 (1933): 758-763.

[HL23] Hardy, G. H., and J. E. Littlewood. "*Partitio numerorum* III: On the Expression of a Number as a Sum of Primes." *Acta Mathematica* 44 (1923): 1-70.

[HP88] Hermann, Mike, and Cameron Patterson. "Sieve Device Announcement." November 22, 1988. University of Calgary, Calgary.

[HP89] Hermann, Mike, and Cameron Patterson. "A High Performance Mathematical Sieve." *Proceedings of the 1989 IEEE Canadian Conference on Electrical and Computer Engineering*. (to appear)

[Kae33] Kaempffert, Waldemar. "'Congruence Machine' Divides Numbers Quickly." *New York Times*, 12 March 1933.

[Knu81] Knuth, Donald E. *Seminumerical Algorithms*. Vol. 2 of *The Art of Computer Programming*. 2nd ed. Reading: Addison-Wesley, 1981.

[Kra22] Kraitchik, Maurice. *Théorie des Nombres*. Vol. 1. Paris: Gauthier-Villars et C^{ie}, 1922.

[Kra24] Kraitchik, Maurice. *Recherches sur la Théorie des Nombres*. Vol. 1. Paris: Gauthier-Villars et C^{ie}, 1924.

[LE39] Lehmer, Derrick Norman. *Factor Stencils*. Revised and extended by John D. Elder. N.p.: Carnegie Institution of Washington, September 1939.

[Leh18] Lehmer, D. N. "On the History of the Problem of Separating a Number into its Prime Factors." *The Scientific Monthly*, September 1918: 227-234.

[Leh28] Lehmer, D. H. "The Mechanical Combination of Linear Forms." *American Mathematical Monthly* 35 (1928): 114-121.

[Leh33a] Lehmer, Derrick N. "Hunting Big Game in the Theory of Numbers." *Scripta Mathematica* (March 1933): 229-235.

[Leh33b] Lehmer, D. H. "A Photo-Electric Number Sieve." *American Mathematical Monthly* 40 (1933): 401-406.

[Leh33c] Lehmer, D. H. "Some New Factorizations of $2^n \pm 1$." *Bulletin of the American Mathematical Society* 39 (1933): 105-108.

[Leh34] Lehmer, D. H. "A Machine for Combining Sets of Linear Congruences." *Mathematische Annalen* 109 (1934): 661-667.

[Leh36] Lehmer, D. H. "On the Function of $x^2 + x + A$." *Sphinx* 6 (1936): 212-214. Also *Sphinx* 7 (1937): 40.

[Leh47] Lehmer, D. H. "On the Factors of $2^n \pm 1$." *Bulletin of the American Mathematical Society* 53 (1947): 164-167.

[Leh49] Lehmer, D. H. "On the Converse of Fermat's Theorem II." *American Mathematical Monthly* 56 (1949): 300-309.

[Leh53] Lehmer, D. H. "The Sieve Problem for All-Purpose Computers." *Mathematical Tables and Other Aids to Computation* 7, no. 41 (1953): 6-14.

[Leh54] Lehmer, D. H. "A Sieve Problem on 'Pseudo-squares'." *Mathematical Tables and Other Aids to Computation* 8, no. 48 (1954): 241-242.

[Leh66] Lehmer, D. H. "An Announcement Concerning the Delay Line SIEVE DLS-127." *Mathematics of Computation* 20 (1966): 645-646.

[Leh68] Lehmer, D. H. "Machines and Pure Mathematics." *Computers in Mathematical Research.* Ed. Churchhouse, R. F., and J.-C. Herz. Amsterdam: North-Holland, 1968.

[Leh74] Lehmer, D. H. "The Influence of Computing on Research in Number Theory." Ed. Joseph P. LaSalle. *The Influence of Computing on Mathematical Research and Education.* Vol 20 of *Proceedings of Symposia in Applied Mathematics.* Providence: American Mathematical Society, 1974. 3-12.

[Leh76] Lehmer, D. H. "Exploitation of Parallelism in Number Theoretic and Combinatorial Computation." Proceedings of 6th Manitoba Conference on Numerical Mathematics. *Congressus Numerantium* 18 (1976): 95-111.

[Leh80] Lehmer, D. H. "A History of the Sieve Process." *A History of Computing in the Twentieth Century.* Academic Press, 1980. 445-456.

[Leh89] Lehmer, D. H. Personal interview. 16 June 1989.

[LL74] Lehmer, D. H., and Emma Lehmer. "A New Factorization Technique Using Quadratic Forms." *Mathematics of Computation* 28 (1974): 625-635.

[LLS70] Lehmer, D. H., and Emma Lehmer, Daniel Shanks. "Integer Sequences Having Prescribed Quadratic Character." *Mathematics of Computation* 24 (1970): 433-451.

[LM78] Lehmer, D. H., and J. M. Masley. "Table of Cyclotomic Class Numbers $h^*(p)$ and Their Factors for $200 < p < 521$." *Mathematics of Computation* 32 (1978): 577-582.

[MA78] Manders, K., and L. Adleman. "NP-Complete Decision Problems for Binary Quadratics." *Jouirnal of Computer and System Sciences* 16 (1978): 168-184.

[MB75] Morrison, Michael A., and John Brillhart. "A Method of Factoring and the Factorization of F_7." *Mathematics of Computation* 29 (1975): 183-205.

[Mot81] Motorola, Inc. *MC6809-MC6809E 8-Bit Microprocessor Programming Manual.* Austin: Motorola, Inc., 1981.

[Mot84] Motorola, Inc. *Single Chip Microcomputer Data.* Austin: Motorola, Inc., 1984.

[MW89] Mollin, R. A., and H. C. Williams. "Quadratic Non-Residues and Prime-Producing Polynomials." *Canadian Mathematical Bulletin* (to appear).

[Oak33] "Wizard Machine Solves Mysteries in Mathematics." *Oakland Tribune*, 15 March 1933.

[Pat83] Patterson, Cameron. "Design and Use of an Electronic Sieve." M.Sc. diss., University of Manitoba, 1983.

[Pat89] Patterson, C. D. "The Complexity of Sieving." Unpublished manuscript, 1989.

[Pom89] Pomerance, Carl. "Factoring." *Introductory Survey Lectures on Cryptology and Computational Number Theory.* American Mathematical Society Short Course Series. Boulder: American Mathematical Society, 1989.

[PW83] Patterson, C. D., and H. C. Williams. "A Report on the University of Manitoba Sieve Unit." *Congressus Numerantium* 37 (1983): 85-98.

[PW85] Patterson, C. D., and H. C. Williams. "Some Periodic Continued Fractions With Long Periods." *Mathematics of Computation* 44 (1985): 523-532.

[Rub83] Rubinstein, Richard. "D. H. Lehmer's Number Sieves." *The Computer Museum Report* (Spring 1983): 2-4.

[Sha59] Shanks, Daniel. "A Sieve Method for Factoring Numbers of the Form n^2+1." *Mathematical Tables and Other Aids to Computation* 13 (1959): 78-86.

[Sha60] Shanks, Daniel. "On the Conjecture of Hardy and Littlewood Concerning the Number of Primes of the Form n^2+a." *Mathematics of Computation* 14 (1960): 320-332.

[Sha63] Shanks, Daniel. "Supplementary Data and Remarks Concerning a Hardy-Littlewood Conjecture." *Mathematics of Computation* 17 (1963): 188-193.

[Sha73] Shanks, Daniel. "Systematic Examination of Littlewood's Bounds on $L(1,\chi)$." Ed. Harold C. Diamond. *Analytic Number Theory*. Vol. 24 of *Proceedings of Symposia in Pure Mathematics*. Providence: American Mathematical Society, 1973. 267-283.

[Ste89] Stephens, Allan J. *mp: A Multi-precise Integer Package*. Winnipeg: Department of Computer Science, University of Manitoba, 1989.

[SW88] Stephens, A. J., and H. C. Williams. "Computation of Real Quadratic Fields with Class Number One." *Mathematics of Computation* 51 (1989): 809-824.

[Wil78] Williams, H. C. "Primality Testing on a Computer." *Ars Combinatoria* 5 (1978): 127-185.

[Wil81] Williams, H. C. "A Numerical Investigation into the Length of the Period of the Continued Fraction Expansion of $\sqrt{D}$." *Mathematics of Computation* 36 (1981): 593-601.

Appendix A : OAS Schematics

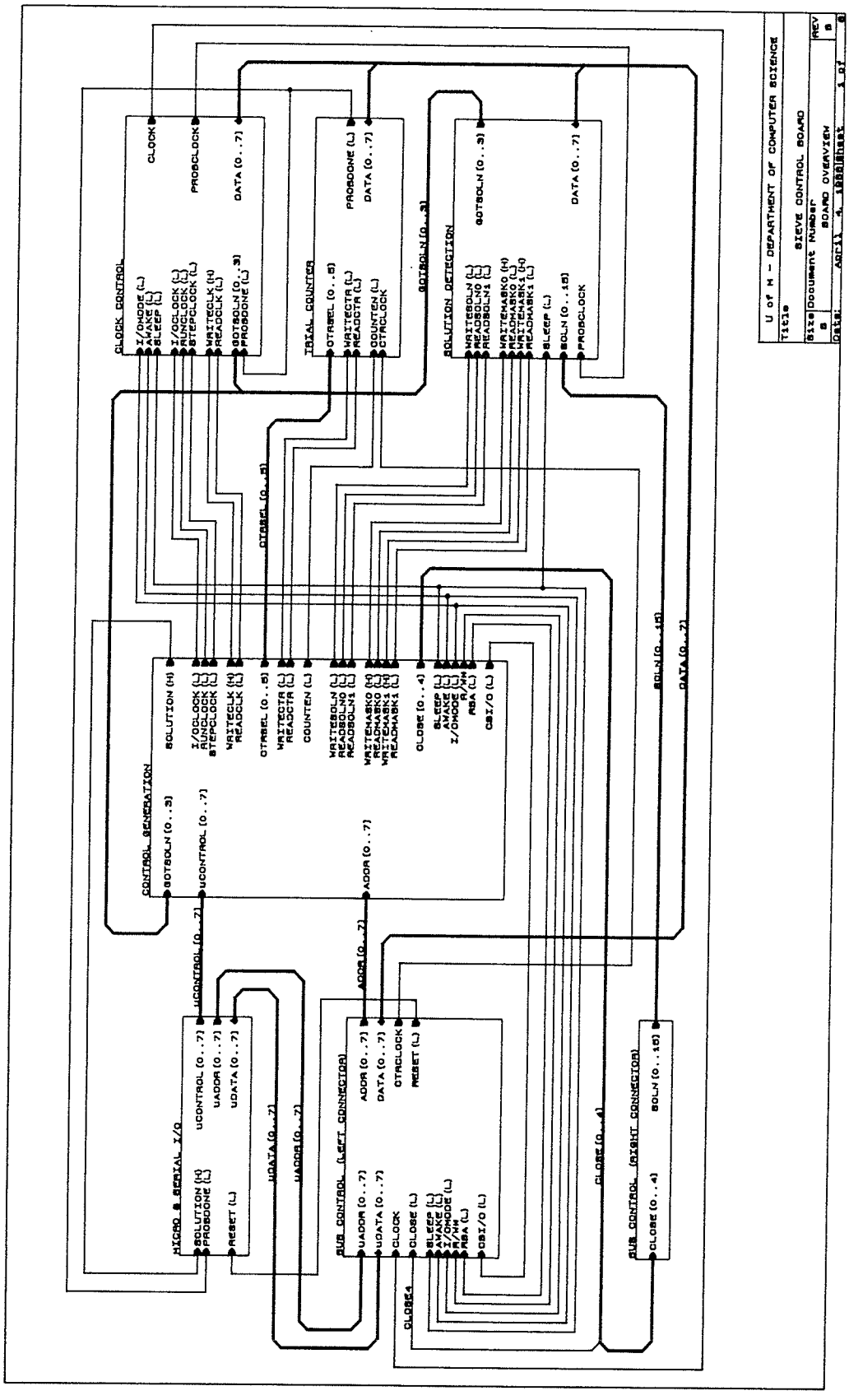
This appendix contains schematic diagrams depicting the design of the control board and ring boards making up an Open Architecture Sieve sieve unit. Together with the PAL definitions in Appendix B, they form a complete description of the OAS hardware at its lowest level and provide an accurate record for future sieve users.

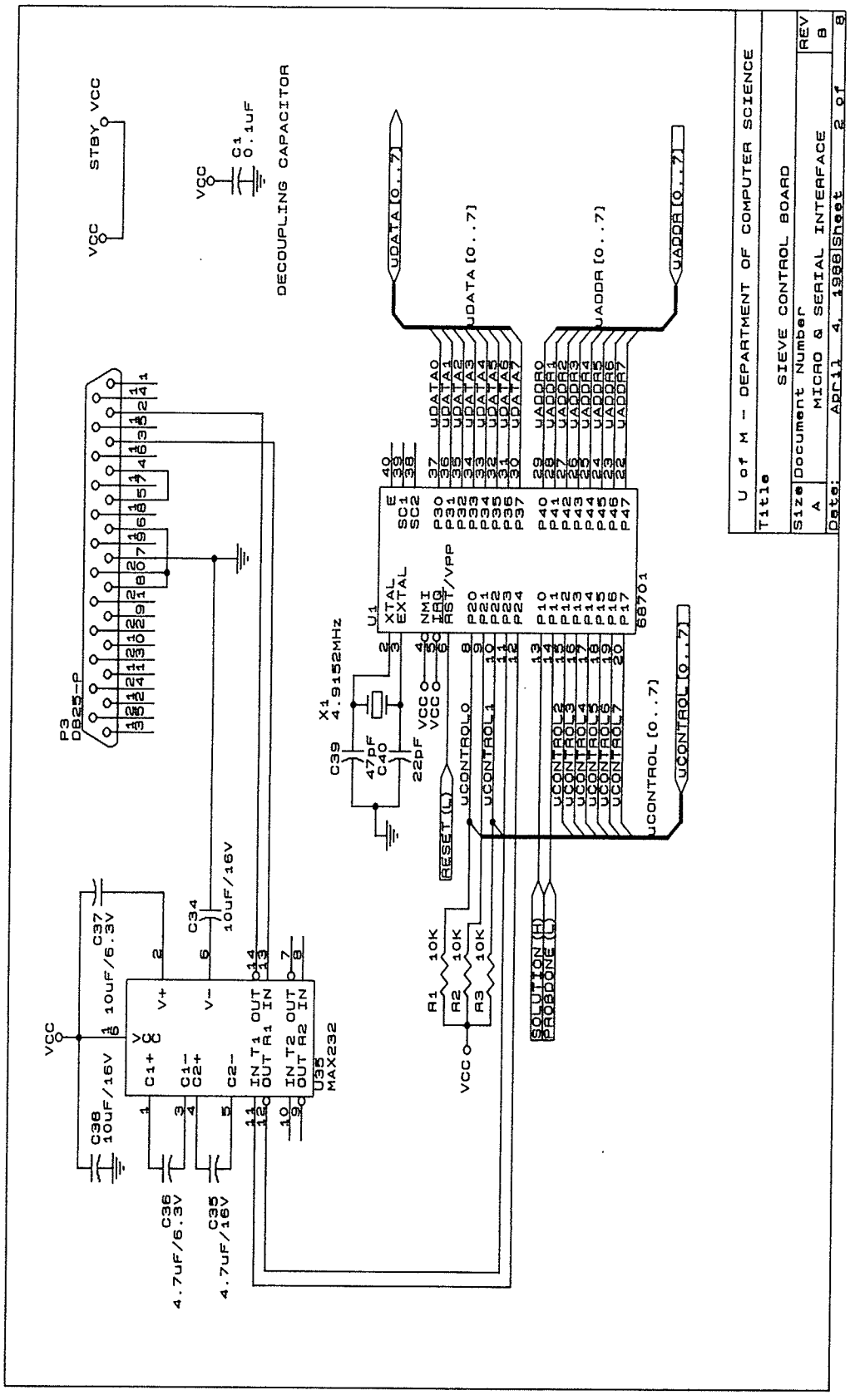
All of the diagrams were produced using the OrCAD/SDT™ schematic design tool package. The first eight diagrams depict the OAS control board, while the final eight depict the ring board. Since all four rings on a ring board are identical in design, pages 6, 7, and 8 of the ring board schematics have been omitted; each missing page is identical to page 5 except for the numbering of its components. Size A worksheets are printed at their normal scale; size B worksheets have been reduced by a factor of two to allow them fit on an 8.5 by 11 inch page.

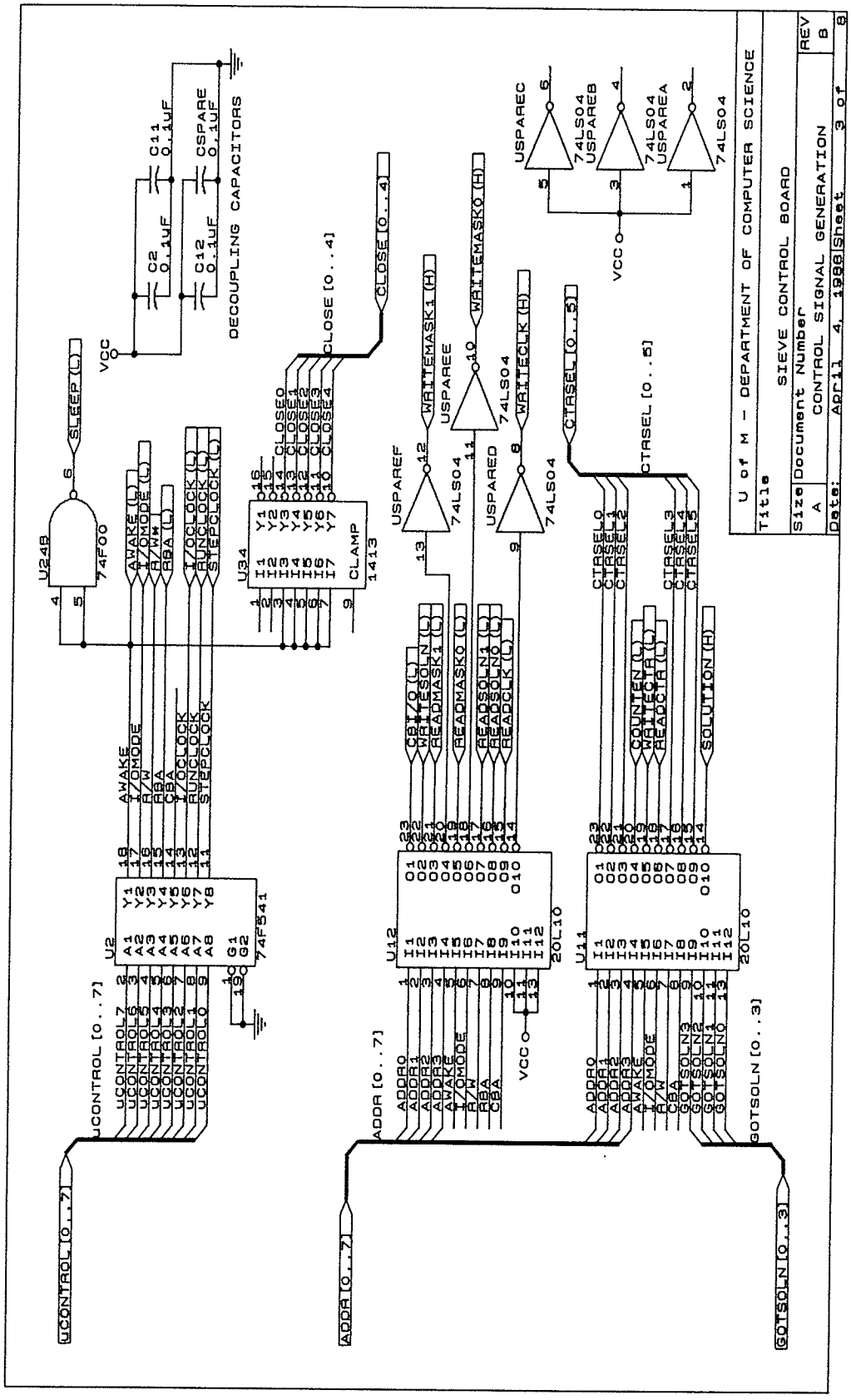
The diagrams show the design of the OAS hardware at the time is was assembled. A number of minor modifications were later made during testing to permit reliable operation. These are noted below.

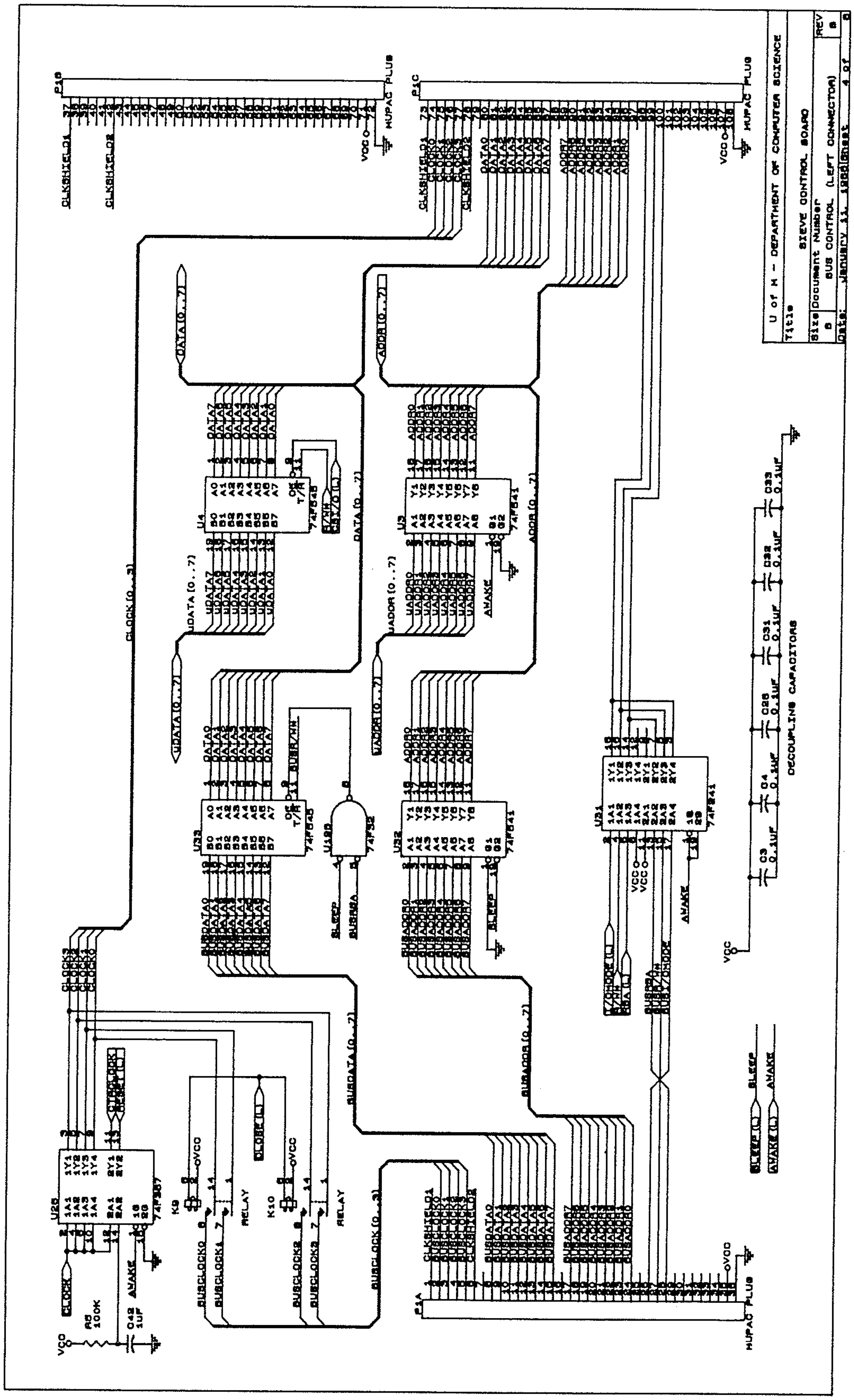
Control Board, page 4	A second 74F367 was mounted on top of U26 to provide better internal grounding for the chip.
	A small switch was inserted in parallel with C42 to allow the RESET (L) signal to be generated manually.
Control Board, page 5	U22 is a 40 MHz oscillator, not 50 MHz as shown.
Control Board, page 8	A second 220Ω resistor pack was tied to RP1 and a second to RP2 to decrease their resistance to 110Ω.
Ring Board, page 5	The clock signal going to each ring is first run through a 68Ω resistor that helps to absorb reflections.
	An additional 0.1μF decoupling capacitor was added to each of the 74F269 counters (U4 and U12).

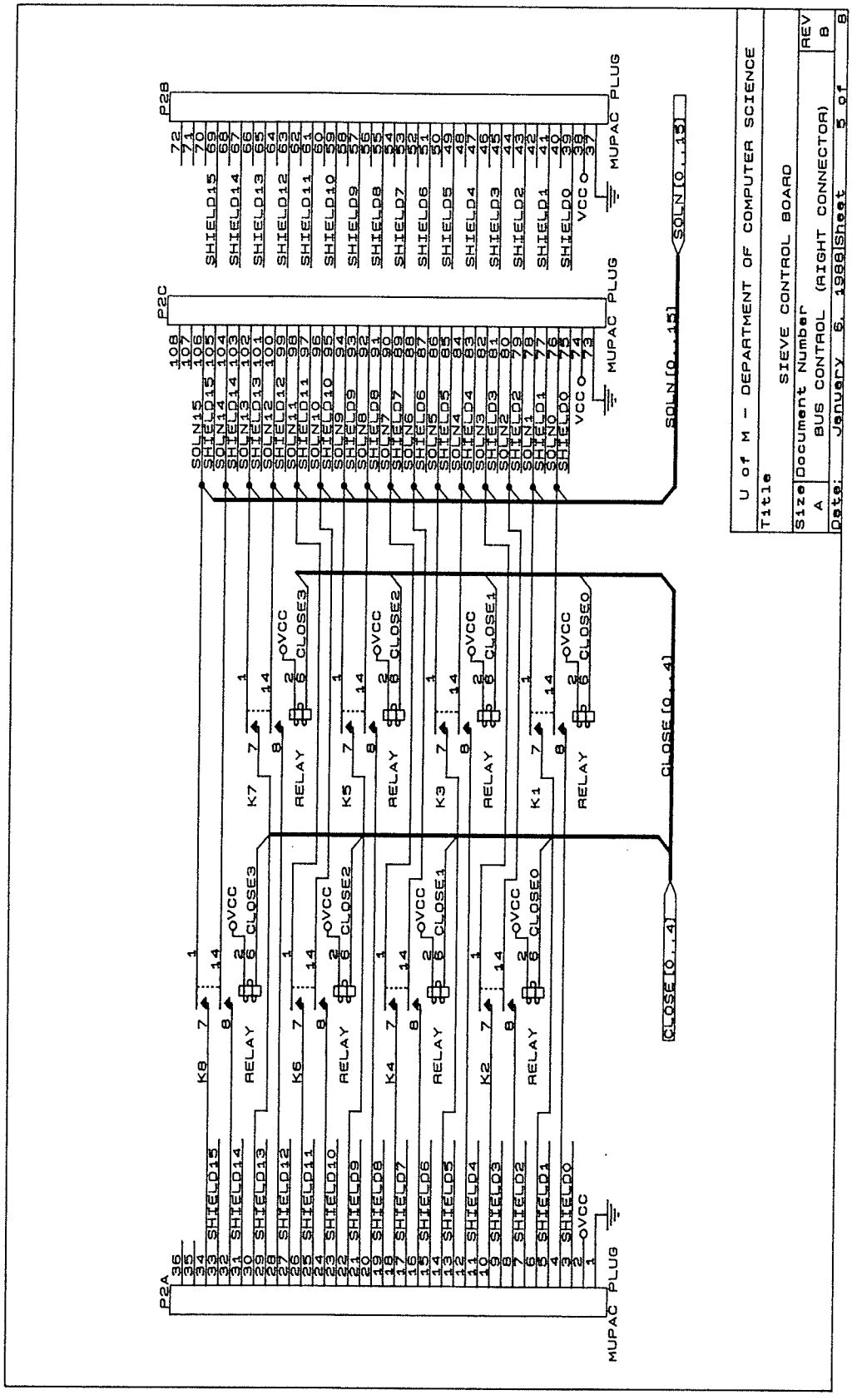
OrCAD/SDT™ is a trademark of OrCAD Systems Corporation.

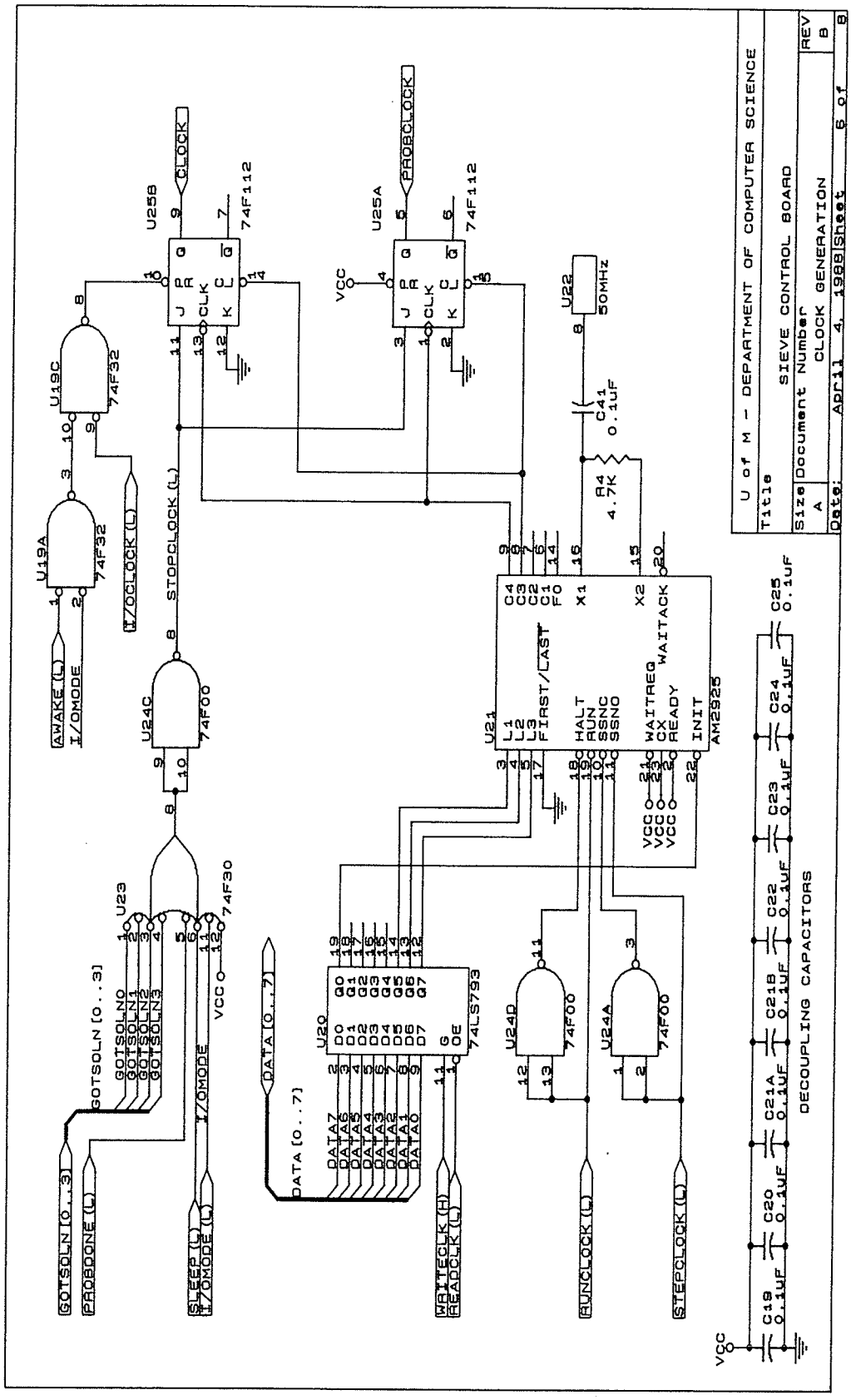

U of M - DEPARTMENT OF COMPUTER SCIENCE
SIEVE CONTROL BOARD
BOARD OVERVIEW
CLOCK CONTROL
TOTAL COUNTER
SOLUTION DETECTION
CONTROL GENERATION
MICRO_8_SERIAL_I/O
SUB CONTROL (LEFT CONNECTION)
SUB CONTROL (RIGHT CONNECTION)

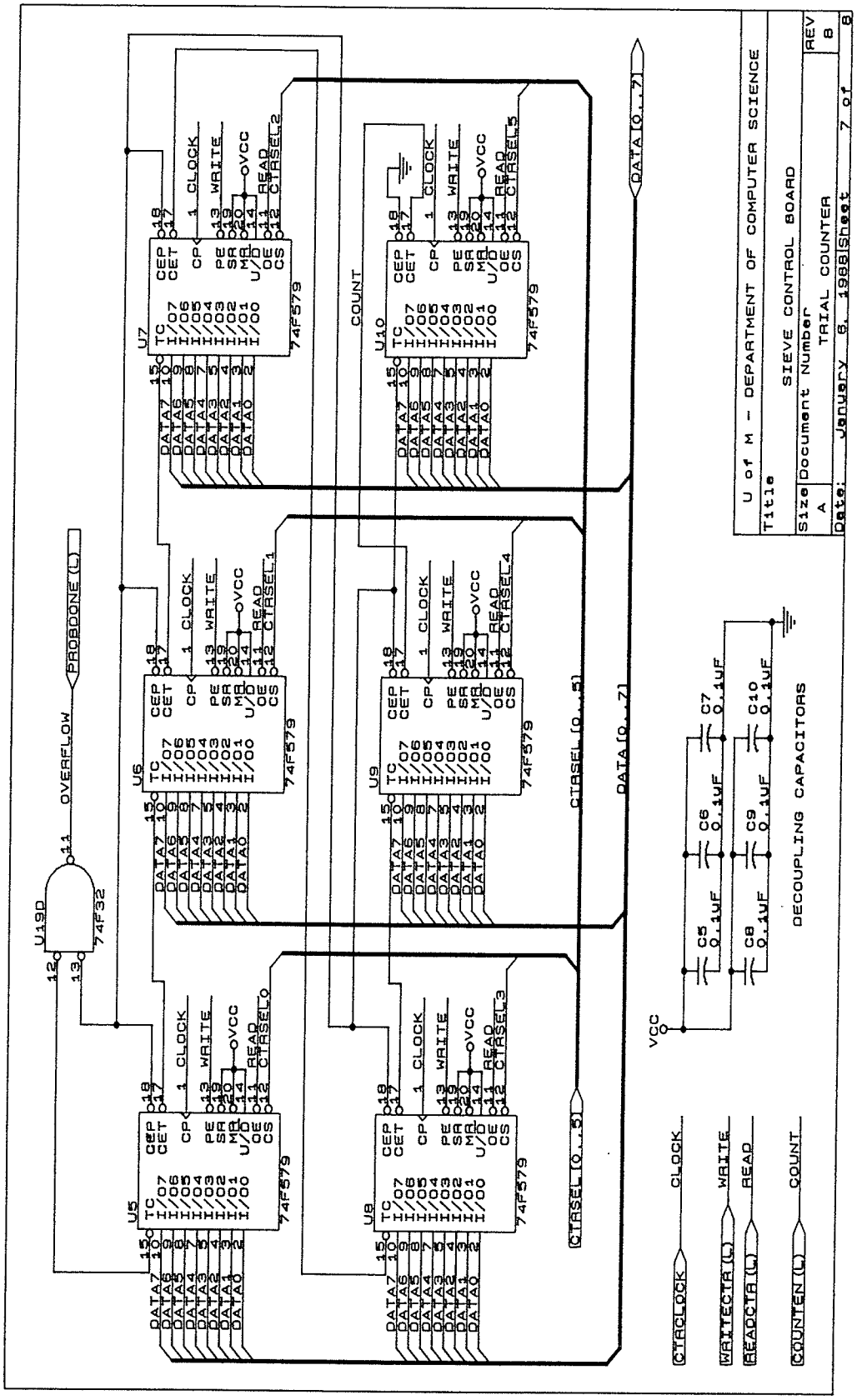

U of M -- DEPARTMENT OF COMPUTER SCIENCE
Title
SIEVE CONTROL BOARD
Size
A
Document Number
MICRO & SERIAL INTERFACE
REV
B
Date:
April 4, 1988
Sheet
2 of 8
DECOUPLING CAPACITOR
VCC
STBY VCC
C1
0.1uF
P3
DB25-P
U1
68701
XTAL
EXTAL
NMI
IRQ
RST/VPP
SC1
SC2
P10
P11
P12
P13
P14
P15
P16
P17
P20
P21
P22
P23
P24
P30
P31
P32
P33
P34
P35
P36
P37
P40
P41
P42
P43
P44
P45
P46
P47
X1
4.9152MHz
C39
47pF
C40
22pF
RESET(L)
UCONTROL0
UCONTROL1
UCONTROL2
UCONTROL3
UCONTROL4
UCONTROL5
UCONTROL6
UCONTROL7
UCONTROL [0..7]
UDATA [0..7]
UADDR [0..7]
UDATA0
UDATA1
UDATA2
UDATA3
UDATA4
UDATA5
UDATA6
UDATA7
UADDR0
UADDR1
UADDR2
UADDR3
UADDR4
UADDR5
UADDR6
UADDR7
R1 10K
R2 10K
R3 10K
SOLUTION(H)
PROBDONE(L)
U35
MAX232
C1+
C1-
C2+
C2-
V+
V-
IN T1 OUT
OUT R1 IN
IN T2 OUT
OUT R2 IN
C34
10uF/16V
C37
10uF/6.3V
C38
10uF/16V
C36
4.7uF/6.3V
C35
4.7uF/16V

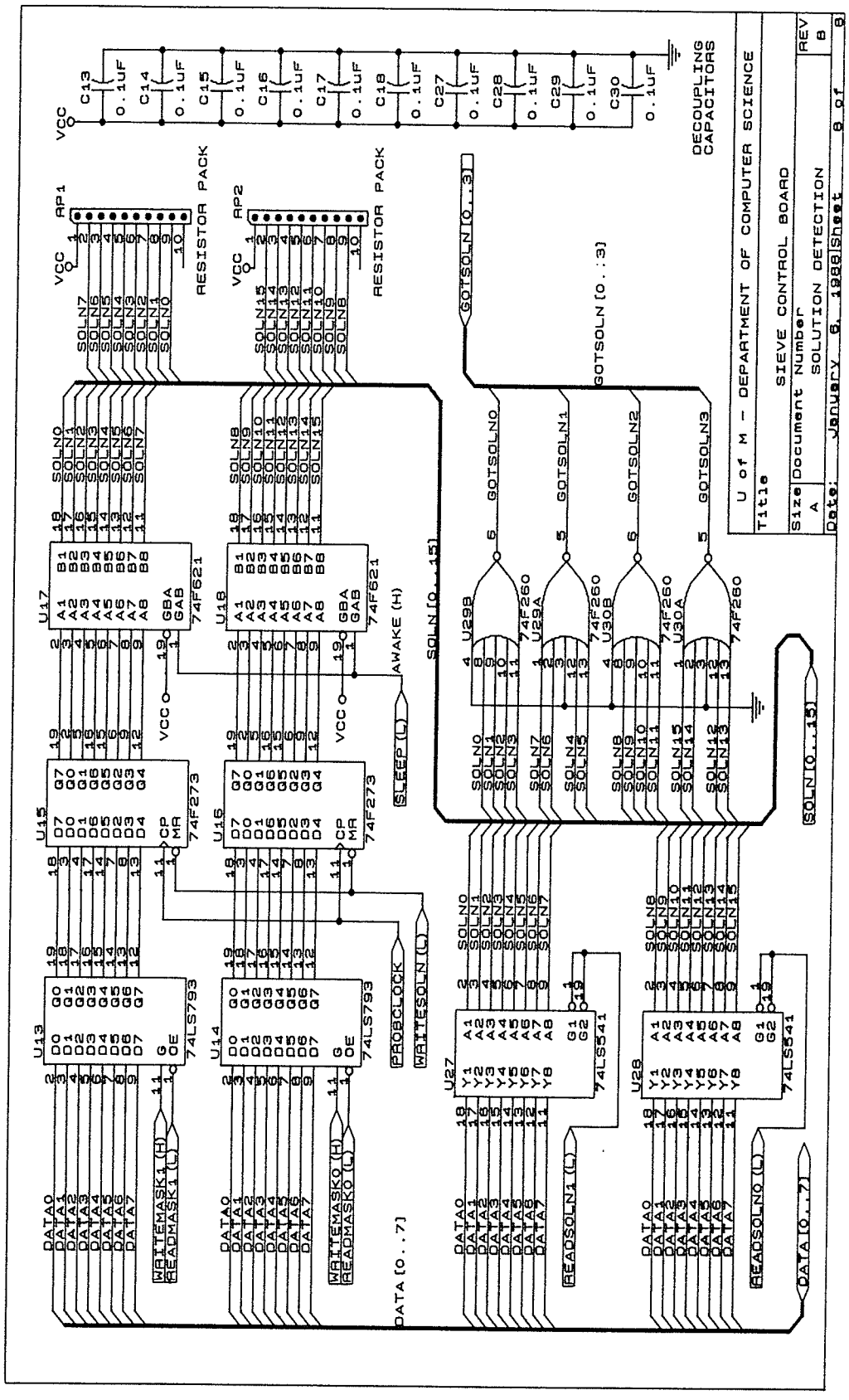

U of M - DEPARTMENT OF COMPUTER SCIENCE
SIEVE CONTROL BOARD
CONTROL SIGNAL GENERATION
DECOUPLING CAPACITORS
C2 0.1uF
C12 0.1uF
C11 0.1uF
CSPARE 0.1uF
VCC
SLEEP (L)
U248
74F00
U2
74F541
U34
CLAMP
1413
U12
U11
20L10
74LS04
UCONTROL [0..7]
ADDR [0..7]
GOTSOLN [0..3]
CLOSE [0..4]
CTRSEL [0..5]
WRITEMASK1 (H)
WRITEMASK0 (H)
WRITECLK (H)
COUNTEN (L)
SOLUTION (H)
USPAREA
USPAREB
USPAREC
USPAREF
USPARED
USPAREE

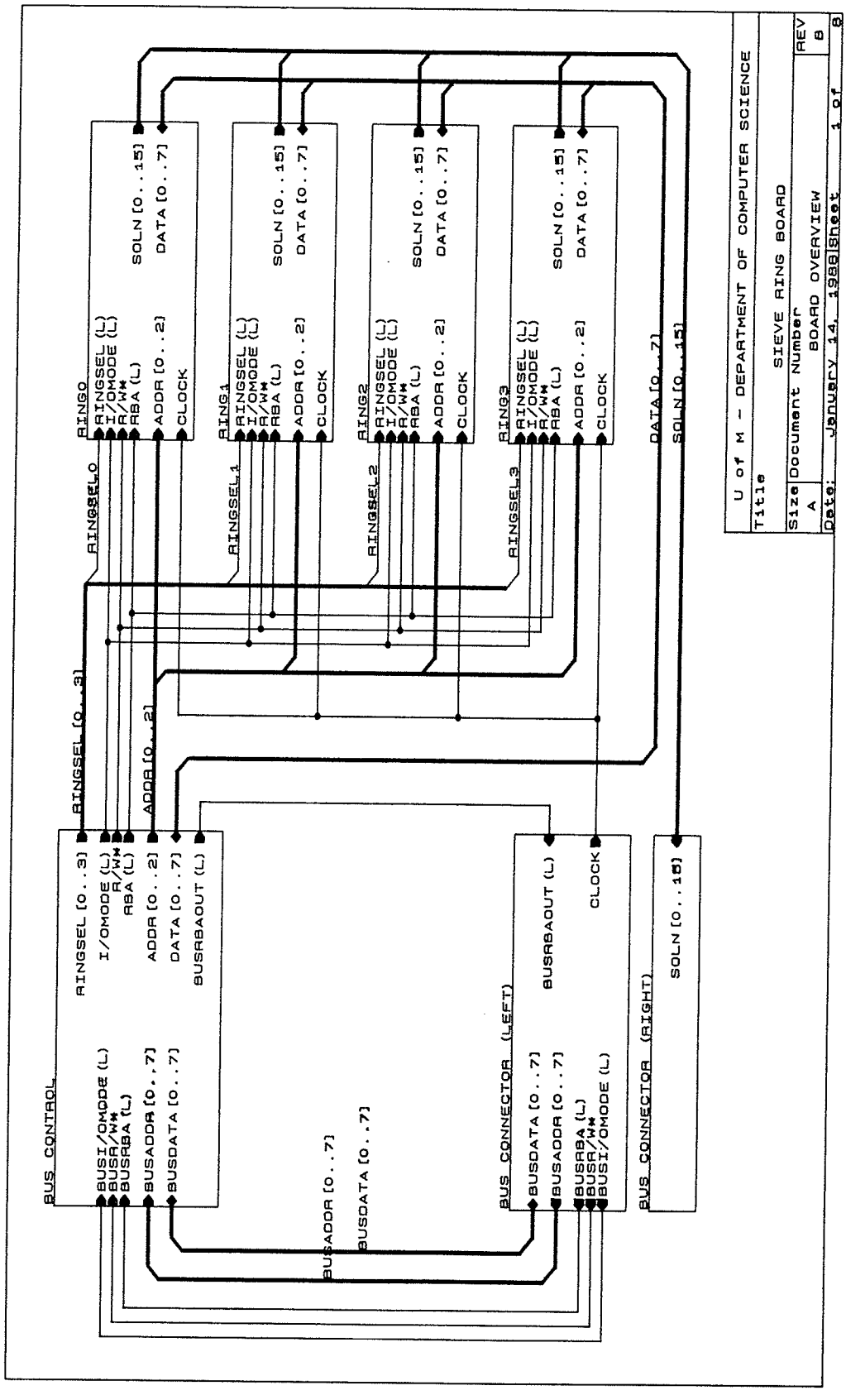

U of M - DEPARTMENT OF COMPUTER SCIENCE
SIEVE CONTROL BOARD
BUS CONTROL (LEFT CONNECTOR)
DECOUPLING CAPACITORS

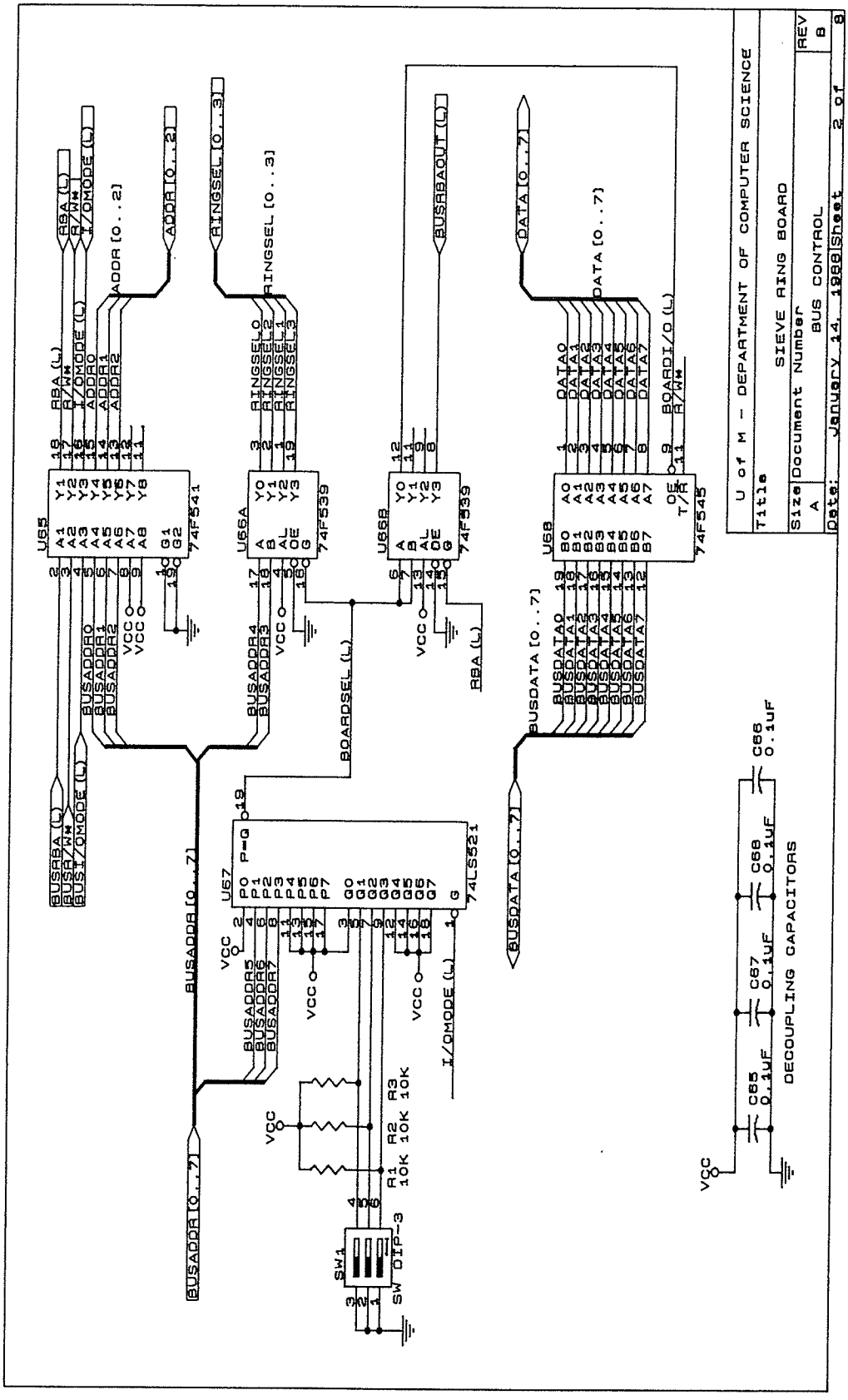

U of M - DEPARTMENT OF COMPUTER SCIENCE
Title
SIEVE CONTROL BOARD
Size
A
Document Number
BUS CONTROL (RIGHT CONNECTOR)
REV
B
Date: January 6, 1988 Sheet 5 of 8
P2A
P2C
P28
MUPAC PLUG
SOLN[0..15]
CLOSE[0..4]
SHIELD15
SHIELD14
SHIELD13
SHIELD12
SHIELD11
SHIELD10
SHIELD9
SHIELD8
SHIELD7
SHIELD6
SHIELD5
SHIELD4
SHIELD3
SHIELD2
SHIELD1
SHIELD0
VCC
K9
K8
K7
K6
K5
K4
K3
K2
K1
RELAY
CLOSE3
CLOSE2
CLOSE1
CLOSE0

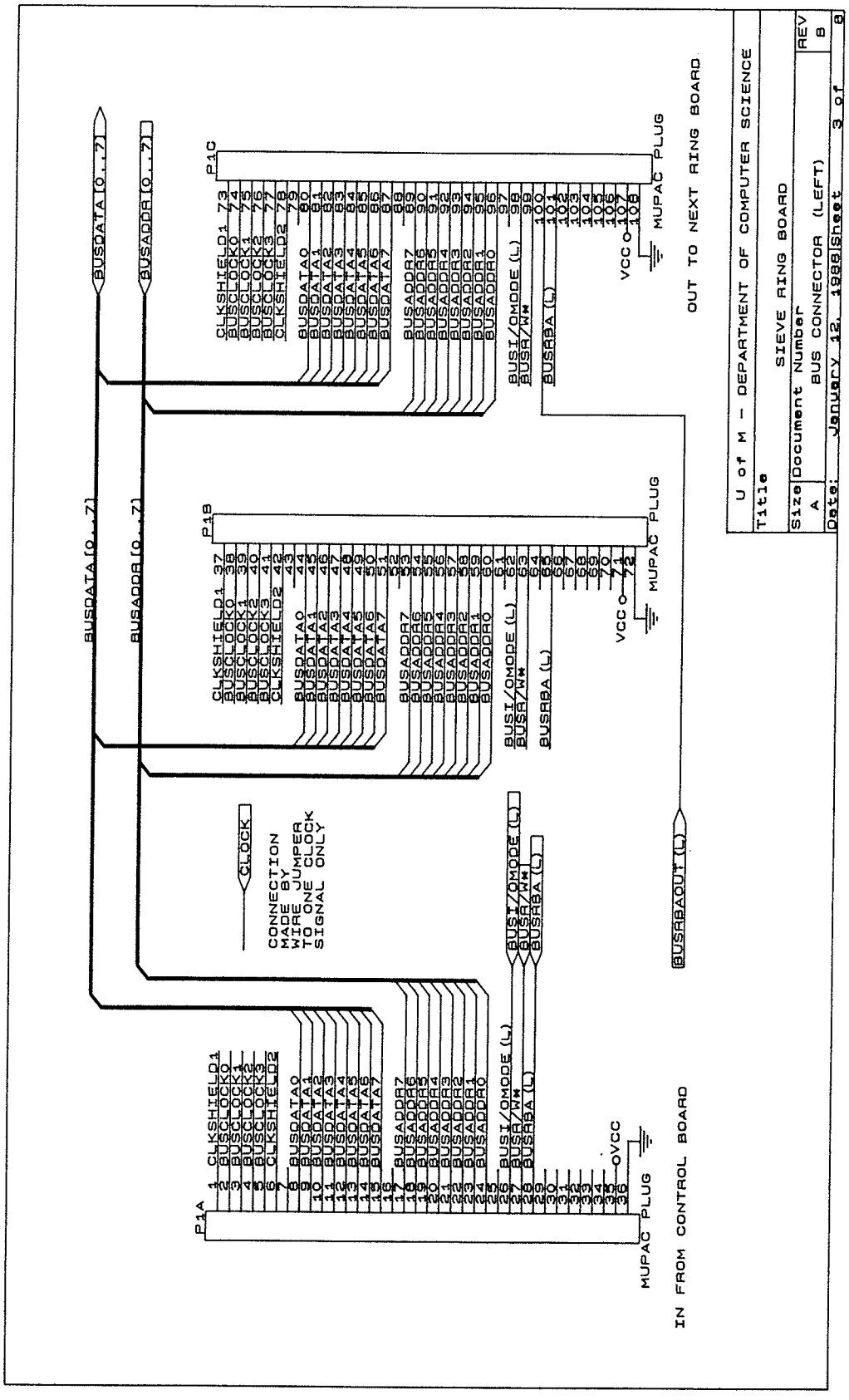

U of M - DEPARTMENT OF COMPUTER SCIENCE
Title
SIEVE CONTROL BOARD
Size A
Document Number
CLOCK GENERATION
REV B
Date: Apr 11, 1988 Sheet 6 of 8
U25B
U25A
74F112
74F32
74F00
74F30
74LS793
AM2925
50MHz
CLOCK
PROBCLOCK
STOPCLOCK (L)
AWAKE (L)
I/OCLOCK (L)
I/OMODE
GOTSOLN [0..3]
PROBDONE (L)
SLEEP (L)
DATA [0..7]
WAITECLK (H)
READCLK (L)
RUNCLOCK (L)
STEPCLOCK (L)
R4 4.7K
C41 0.1uF
DECOUPLING CAPACITORS

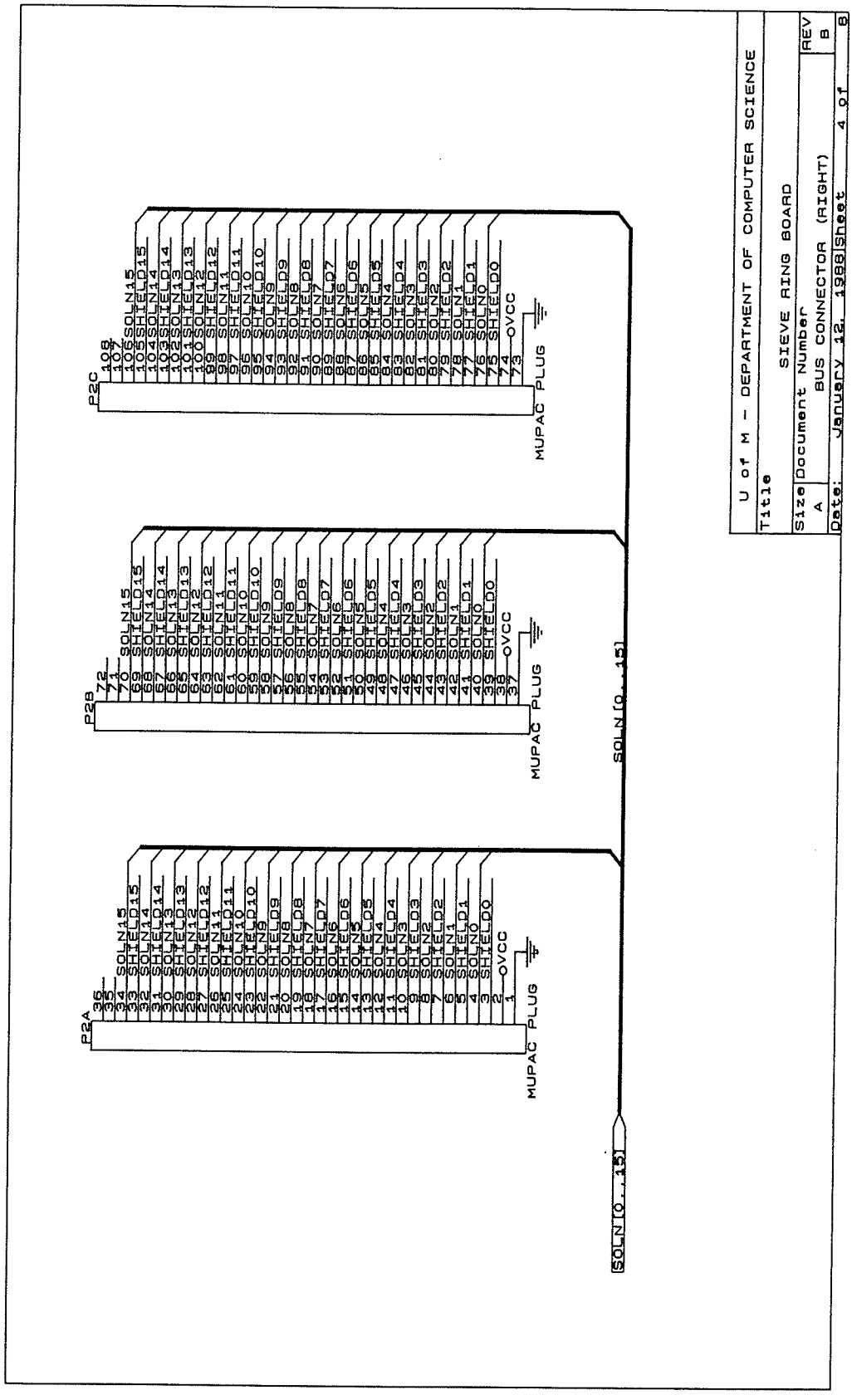

U of M - DEPARTMENT OF COMPUTER SCIENCE
Title
SIEVE CONTROL BOARD
Size
A
Document Number
TRIAL COUNTER
REV
8
Date: January 6, 1988
Sheet 7 of 8
U19D
74F32
OVERFLOW
PROBDONE(L)
U5
U6
U7
U8
U9
U10
74F579
DATA7
DATA6
DATA5
DATA4
DATA3
DATA2
DATA1
DATA0
CEP
CET
CP
PE
SR
M0
U/D
OE
CS
TC
I/O7
I/O6
I/O5
I/O4
I/O3
I/O2
I/O1
I/O0
CLOCK
WRITE
READ
CTRSEL0
CTRSEL1
CTRSEL2
CTRSEL3
CTRSEL4
CTRSEL5
COUNT
CTRSEL(0..5)
DATA(0..7)
C5
C6
C7
C8
C9
C10
0.1uF
DECOUPLING CAPACITORS
VCC
CTRCLOCK
WRITECTR(L)
READCTR(L)
COUNTEN(L)

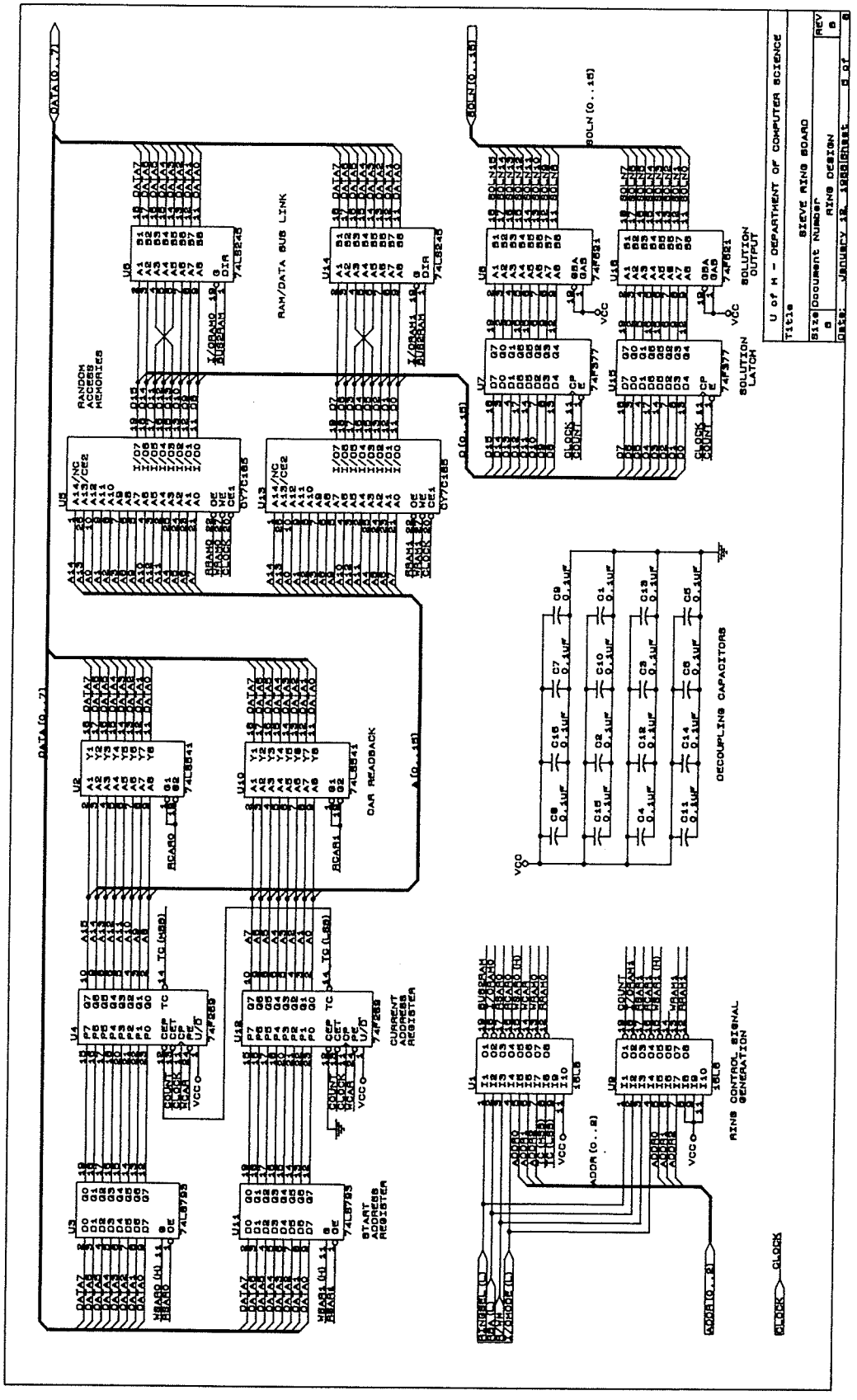

U of M - DEPARTMENT OF COMPUTER SCIENCE
Title
SIEVE CONTROL BOARD
Size A
Document Number
SOLUTION DETECTION
REV B
Date: January 6, 1988 Sheet 8 of 8
DECOUPLING CAPACITORS
C13 0.1uF
C14 0.1uF
C15 0.1uF
C16 0.1uF
C17 0.1uF
C18 0.1uF
C27 0.1uF
C28 0.1uF
C29 0.1uF
C30 0.1uF
VCC
RP1
RP2
RESISTOR PACK
U13 74LS793
U14 74LS793
U15 74F273
U16 74F273
U17 74F621
U18 74F621
U27 74LS541
U28 74LS541
U29A 74F260
U30A 74F260
U29B 74F260
U30B 74F260
GOTSOLN[0..3]
GOTSOLN0
GOTSOLN1
GOTSOLN2
GOTSOLN3
SOLN[0..15]
AWAKE (H)
SLEEP (L)
PROBCLOCK
WRITESOLN (L)
READSOLN (L)
READSOLN0 (L)
DATA[0..7]
WRITEMASK1 (H)
READMASK1 (L)
WRITEMASK0 (H)
READMASK0 (L)


U of M - DEPARTMENT OF COMPUTER SCIENCE
Title
SIEVE RING BOARD
Size
A
Document Number
BOARD OVERVIEW
REV
B
Date: January 14, 1988 Sheet 1 of 8
RING0
RING1
RING2
RING3
RINGSEL (L)
I/OMODE (L)
R/W*
RBA (L)
ADDR [0..2]
CLOCK
SOLN [0..15]
DATA [0..7]
RINGSEL0
RINGSEL1
RINGSEL2
RINGSEL3
RINGSEL [0..3]
ADDR [0..2]
DATA [0..7]
SOLN [0..15]
BUS CONTROL
RINGSEL [0..3]
I/OMODE (L)
R/W*
RBA (L)
ADDR [0..2]
DATA [0..7]
BUSRBAOUT (L)
BUSI/OMODE (L)
BUSR/W*
BUSRBA (L)
BUSADDR [0..7]
BUSDATA [0..7]
BUS CONNECTOR (LEFT)
BUSDATA [0..7]
BUSADDR [0..7]
BUSRBA (L)
BUSR/W*
BUSI/OMODE (L)
BUSRBAOUT (L)
CLOCK
BUS CONNECTOR (RIGHT)
SOLN [0..15]


U of M -- DEPARTMENT OF COMPUTER SCIENCE
Title
SIEVE RING BOARD
Size A
Document Number
BUS CONTROL
REV B
Date: January 14, 1988 Sheet 2 of 8
U65
74F541
U66A
74F539
U66B
74F539
U67
74LS521
U68
74F545
SW1
SW DIP-3
R1 10K
R2 10K
R3 10K
C65 0.1uF
C66 0.1uF
C67 0.1uF
C68 0.1uF
DECOUPLING CAPACITORS
BUSADDR [0..7]
BUSDATA [0..7]
ADDR [0..2]
RINGSEL [0..3]
DATA [0..7]
BUSRBA (L)
BUSR/W
BUSI/OMODE (L)
BUSRBAOUT (L)
BOARDSEL (L)
RBA (L)
R/W
I/OMODE (L)
BOARDI/O (L)
VCC


U of M - DEPARTMENT OF COMPUTER SCIENCE
Title
SIEVE RING BOARD
Size
A
Document Number
BUS CONNECTOR (LEFT)
REV
B
Date: January 12, 1988 Sheet 3 of 8
IN FROM CONTROL BOARD
OUT TO NEXT RING BOARD
MUPAC PLUG
P1A
P1B
P1C
CLOCK
CONNECTION MADE BY WIRE JUMPER TO ONE CLOCK SIGNAL ONLY
BUSDATA[0..7]
BUSADDR[0..7]
BUSRBAOUT(L)
CLKSHIELD1
BUSCLOCK0
BUSCLOCK1
BUSCLOCK2
BUSCLOCK3
CLKSHIELD2
BUSDATA0
BUSDATA1
BUSDATA2
BUSDATA3
BUSDATA4
BUSDATA5
BUSDATA6
BUSDATA7
BUSADDR7
BUSADDR6
BUSADDR5
BUSADDR4
BUSADDR3
BUSADDR2
BUSADDR1
BUSADDR0
BUSI/OMODE(L)
BUSR/WM
BUSRBA(L)
VCC
0VCC


U of M - DEPARTMENT OF COMPUTER SCIENCE
Title
SIEVE RING BOARD
Size
A
Document Number
BUS CONNECTOR (RIGHT)
REV
B
Date: January 12, 1988 Sheet 4 of 8
P2A
P2B
P2C
MUPAC PLUG
SOLN[0..15]
SOLN15
SHIELD15
SOLN14
SHIELD14
SOLN13
SHIELD13
SOLN12
SHIELD12
SOLN11
SHIELD11
SOLN10
SHIELD10
SOLN9
SHIELD9
SOLN8
SHIELD8
SOLN7
SHIELD7
SOLN6
SHIELD6
SOLN5
SHIELD5
SOLN4
SHIELD4
SOLN3
SHIELD3
SOLN2
SHIELD2
SOLN1
SHIELD1
SOLN0
SHIELD0
+VCC


U of M - DEPARTMENT OF COMPUTER SCIENCE
SIEVE RING BOARD
RING DESIGN
RANDOM ACCESS MEMORIES
RAM/DATA BUS LINK
SOLUTION LATCH
SOLUTION OUTPUT
DAR READBACK
CURRENT ADDRESS REGISTER
START ADDRESS REGISTER
DECOUPLING CAPACITORS
RING CONTROL SIGNAL GENERATION
DATA(0..7)
A(0..15)
SOLN(0..15)
ADDR(0..2)
CLOCK

Appendix B : OAS PAL Descriptions

B.1 PAL Usage in the Open Architecture Sieve

Programmable Array Logic (PAL®) chips are used in both the OAS control board and ring board designs. In both cases, the primary function of the PALs is to help carry out the control board microcomputer's read and write requests by asserting the appropriate chip input(s) for the correct sieve component. The PALs also generate signals which prepare selected sieve components for sieving when the sieve is placed in problem mode and others which inform the microcomputer when a solution has been found during sieving. Using PALs to carry out these functions, instead of a network of demultiplexers, results in a considerable reduction in board space and complexity.

This appendix describes the internal operation of the two PALs found on the OAS control board and the two PALs used by each ring on the ring boards. Each PAL is represented in two ways: by a set of Boolean equations and by the corresponding fuse map. The way in which the OAS PALs are connected to the other sieve components can be found by examining the schematic diagrams of Appendix A.

B.2 Reading PALASM Equations

The Boolean equations describing the operation of the OAS PALs are written in PALASM® A complete description of this language can be found in the handbook *Programmable Array Logic* (Advanced Micro Devices, Inc., 1983), but the following summary should be sufficient to allow the OAS PAL equations to be interpreted easily.

A PALASM program file consists of three parts: a *documentation section*, a *pin description section*, and an *output signal section.*

The documentation section is contained in the first three lines of the file. It specifies the type of PAL being programmed and describes the project the PAL is used in.

PAL® and PALASM® are registered trademarks of Monolithic Memories, Inc.

The pin description section assigns a name to each of the PALs pins, beginning with pin 1. A name prefixed by a slash ('/') indicates that the pin is considered active LOW whenever it is used as an input, rather than the default of active HIGH. It is irrelevant whether the name of an output pin is preceded by a slash or not—the output is active HIGH or active LOW according to the type of PAL being programmed.

The output signal section consists of a series of Boolean equations that define the conditions under which each output pin is asserted. Each equation contains an output pin name, and equals sign ('='), and an expression representing the conditions under which the output is asserted. The simplest expression is the name of an input pin, which indicates that the output is asserted whenever the input is. An input name prefixed by a slash causes the output to be asserted when the input is *de*asserted. More complex expressions can be formed by logically ANDing two or more terms using asterisks ('*') or ORing them using plus signs ('+'); if both operations are used in combination, the AND operator has higher priority. All characters from a semicolon (';') up to the end of the line signify a comment.

The PALASM language provides more complex features to program chips which have internal flip-flops, *bidirectional* pins (a pin that can be used for both input and output), or *tri-stateable* outputs (a pin which can be temporarily disconnected from its normal output source within the PAL). However, since none of the OAS PALs has these capabilities, a description of how these features are represented in PALASM need not be given.

B.3 Reading PALASM Fuse Maps

Once a PALASM description of a PAL has been created, it is converted to the corresponding fuse map by running the text through a PALASM assembler. The fuse map is then read by PAL programming software which blows the appropriate fuses in the PAL. The PAL designer can use a fuse map to manually verify that the PALASM equations have been correctly written and translated before actually programming a PAL chip.

No fuse map can be easily interpreted without a diagram showing the connections between the PAL's pins and the fuse grid. Such diagrams can be found in the *PAL® Device Data Book* (Advanced Micro Devices, Inc., 1988) and similar sources. In the fuse maps that follow, a '0' indicates an intact fuse (ie. grid connection) while a '1' indicates a blown fuse (ie. no connection).

PAL EQUATIONS - CONTROL BOARD PAL0

```
PAL20L10
ALLAN STEPHENS  4/APRIL/88
CONTROL SIGNALS FOR CONTROL BOARD TRIAL COUNTER & SOLUTION DETECTION

A0          A1          A2          A3          /AWAKE     /IOMODE
RW          /CBA        /GOTSOLN3   /GOTSOLN2   /GOTSOLN1  GND
/GOTSOLN0   SOLUTION    /CTRSEL5    /CTRSEL4    /CTRSEL3   /READCTR
/WRITECTR   /COUNTEN    /CTRSEL2    /CTRSEL1    /CTRSEL0   VCC

CTRSEL0    = AWAKE*IOMODE*A3*/A2*A1*/A0  ; counter segment selects
CTRSEL1    = AWAKE*IOMODE*A3*/A2*A1*A0
CTRSEL2    = AWAKE*IOMODE*A3*A2*/A1*/A0
CTRSEL3    = AWAKE*IOMODE*A3*A2*/A1*A0
CTRSEL4    = AWAKE*IOMODE*A3*A2*A1*/A0
CTRSEL5    = AWAKE*IOMODE*A3*A2*A1*A0
READCTR    = AWAKE*IOMODE*RW*CBA         ; counter IO only in IO mode
WRITECTR   = AWAKE*IOMODE*/RW*CBA
COUNTEN    = AWAKE*/IOMODE               ; counting in problem mode
; active HIGH signal
/SOLUTION = /AWAKE                       ; solution only if awake
          + /GOTSOLN0*/GOTSOLN1*/GOTSOLN2*/GOTSOLN3
```

FUSE MAP - CONTROL BOARD PAL0

```
(C) Copyright 1987 MicroWay, Inc. All Rights Reserved.
PAL20L10
ALLAN STEPHENS  4/APRIL/88
CONTROL SIGNALS FOR CONTROL BOARD TRIAL COUNTER & SOLUTION DETECTION

*D2206*F0*
L0000 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 *
L0040 0110 1011 0111 1011 1011 1111 1111 1111 1111 1111 *
L0160 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 *
L0200 0101 1011 0111 1011 1011 1111 1111 1111 1111 1111 *
L0320 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 *
L0360 1010 0111 0111 1011 1011 1111 1111 1111 1111 1111 *
L0480 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 *
L0520 1111 1111 1111 1011 0111 1111 1111 1111 1111 1111 *
L0640 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 *
L0680 1111 1111 1111 1011 1011 1011 1011 1111 1111 1111 *
L0800 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 *
L0840 1111 1111 1111 1011 1011 0111 1011 1111 1111 1111 *
L0960 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 *
L1000 1001 0111 0111 1011 1011 1111 1111 1111 1111 1111 *
L1120 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 *
L1160 0110 0111 0111 1011 1011 1111 1111 1111 1111 1111 *
L1280 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 *
L1320 0101 0111 0111 1011 1011 1111 1111 1111 1111 1111 *
L1440 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 *
L1480 1111 1111 1111 0111 1111 1111 1111 1111 1111 1111 *
L1520 1111 1111 1111 1111 1111 1111 1111 0111 0111 0101 *
DB89
```

PAL EQUATIONS - CONTROL BOARD PAL1

```
PAL20L10
ALLAN STEPHENS  4/APRIL/88
CONTROL SIGNALS FOR CONTROL BOARD

A0         A1         A2         A3         /AWAKE     /IOMODE
RW         /RBA       /CBA       P10        P11        GND
P13        /WRITECLK  /READCLK   /READSOL0  /READSOL1  /WRITMSK0
/READMSK0  /WRITMSK1  /READMSK1  /WRITESOL  /CBIO      VCC

READCLK   = AWAKE*IOMODE*RW*CBA*/A3*/A2*/A1*A0  ; R/W clock control
WRITECLK  = AWAKE*IOMODE*/RW*CBA*/A3*/A2*/A1*A0
READMSK0  = AWAKE*IOMODE*RW*CBA*/A3*/A2*A1*/A0  ; IO soln bus mask MSB
WRITMSK0  = AWAKE*IOMODE*/RW*CBA*/A3*/A2*A1*/A0
READMSK1  = AWAKE*IOMODE*RW*CBA*/A3*/A2*A1*A0   ; IO soln bus mask LSB
WRITMSK1  = AWAKE*IOMODE*/RW*CBA*/A3*/A2*A1*A0
READSOL0  = AWAKE*IOMODE*RW*CBA*/A3*A2*/A1*/A0  ; read solution bus MSB
READSOL1  = AWAKE*IOMODE*RW*CBA*/A3*A2*/A1*A0   ; read solution bus LSB
WRITESOL  = AWAKE*IOMODE*/RW*CBA*/A3*A2*/A1      ; clear entire soln bus
CBIO      = AWAKE*IOMODE*CBA                     ; enable transceiver to
          + AWAKE*IOMODE*RBA                     ; microcomputer data bus
```

FUSE MAP - CONTROL BOARD PAL1


```
PAL20L10
ALLAN STEPHENS  4/APRIL/88
CONTROL SIGNALS FOR CONTROL BOARD

*D2206*F0*
L0000 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 *
L0040 1111 1111 1111 1011 1011 1111 1111 1011 1111 1111 *
L0080 1111 1111 1111 1011 1011 1111 1011 1111 1111 1111 *
L0160 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 *
L0200 1011 0111 1011 1011 1011 1011 1111 1011 1111 1111 *
L0320 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 *
L0360 0101 1011 1011 1011 1011 0111 1111 1011 1111 1111 *
L0480 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 *
L0520 0101 1011 1011 1011 1011 1011 1111 1011 1111 1111 *
L0640 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 *
L0680 0110 1011 1011 1011 1011 0111 1111 1011 1111 1111 *
L0800 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 *
L0840 0110 1011 1011 1011 1011 1011 1111 1011 1111 1111 *
L0960 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 *
L1000 1001 0111 1011 1011 1011 0111 1111 1011 1111 1111 *
L1120 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 *
L1160 1010 0111 1011 1011 1011 0111 1111 1011 1111 1111 *
L1280 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 *
L1320 1001 1011 1011 1011 1011 0111 1111 1011 1111 1111 *
L1440 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 *
L1480 1001 1011 1011 1011 1011 1011 1111 1011 1111 1111 *
DB6F
```

PAL EQUATIONS - RING BOARD PAL0

```
PAL16L8
ALLAN STEPHENS  22/APRIL/88
CONTROL SIGNALS FOR MOST SIGNIFICANT HALF OF RING

/RINGSEL /RBA RW /IOMODE A0 A1 A2 /TC0 /TC1 GND
P11 /RRAM0 /WRAM0 /WCAR WSAR0 /RCAR0 /RSAR0 /IORAM0 /BUS2RAM VCC

BUS2RAM = /RW                                  ; RAM IO direction
IORAM0  = RINGSEL*RBA*A2*/A1*/A0               ; do RAM IO on read or write
RSAR0   = RINGSEL*RBA*RW*/A2*A1*/A0            ; read SAR
RCAR0   = RINGSEL*RBA*RW*A2*A1*/A0             ; read CAR
/WSAR0  = /RINGSEL + /RBA + RW + A2 + /A1 + A0  ; write SAR
        ; RINGSEL*RBA*/RW*/A2*A1*/A0 inverted, since active HIGH signal
WCAR    = RINGSEL*RBA*/RW*A2*A1 ; load on write to either CAR byte
        + TC0*TC1                ; or on overflow while counting
WRAM0   = RINGSEL*RBA*/RW*A2*/A1*/A0           ; write RAM during IO only
RRAM0   = RINGSEL*RBA*RW*A2*/A1*/A0            ; read RAM normally
        + /IOMODE                              ; or in problem mode
```

FUSE MAP - RING BOARD PAL0

```
(C) Copyright 1987 MicroWay, Inc. All Rights Reserved.
PAL16L8
ALLAN STEPHENS  22/APRIL/88
CONTROL SIGNALS FOR MOST SIGNIFICANT HALF OF RING

*D2217*F0*
L0000 1111 1111 1111 1111 1111 1111 1111 1111 *
L0032 1111 1011 1111 1111 1111 1111 1111 1111 *
L0256 1111 1111 1111 1111 1111 1111 1111 1111 *
L0288 1010 1111 1111 1011 1011 0111 1111 1111 *
L0512 1111 1111 1111 1111 1111 1111 1111 1111 *
L0544 1010 0111 1111 1011 0111 1011 1111 1111 *
L0768 1111 1111 1111 1111 1111 1111 1111 1111 *
L0800 1010 0111 1111 1011 0111 0111 1111 1111 *
L1024 1111 1111 1111 1111 1111 1111 1111 1111 *
L1056 1101 1111 1111 1111 1111 1111 1111 1111 *
L1088 0111 1111 1111 1111 1111 1111 1111 1111 *
L1120 1111 0111 1111 1111 1111 1111 1111 1111 *
L1152 1111 1111 1111 1111 1111 0111 1111 1111 *
L1184 1111 1111 1111 1111 1011 1111 1111 1111 *
L1216 1111 1111 1111 0111 1111 1111 1111 1111 *
L1280 1111 1111 1111 1111 1111 1111 1111 1111 *
L1312 1010 1011 1111 1111 0111 0111 1111 1111 *
L1344 1111 1111 1111 1111 1111 1111 1011 1011 *
L1536 1111 1111 1111 1111 1111 1111 1111 1111 *
L1568 1010 1011 1111 1011 1011 0111 1111 1111 *
L1792 1111 1111 1111 1111 1111 1111 1111 1111 *
L1824 1010 0111 1111 1011 1011 0111 1111 1111 *
L1856 1111 1111 0111 1111 1111 1111 1111 1111 *
C7B5
```

PAL EQUATIONS - RING BOARD PAL1

```
PAL16L8
ALLAN STEPHENS  8/MARCH/88
CONTROL SIGNALS FOR LEAST SIGNIFICANT HALF OF RING

/RINGSEL /RBA RW /IOMODE A0 A1 A2 P8 P9 GND
P11 /RRAM1 /WRAM1 P14 WSAR1 /RCAR1 /RSAR1 /IORAM1 /COUNT VCC

COUNT   = /IOMODE                          ; count during problem mode
IORAM1  = RINGSEL*RBA*A2*/A1*A0            ; do RAM IO on read or write
RSAR1   = RINGSEL*RBA*RW*/A2*A1*A0         ; read SAR
RCAR1   = RINGSEL*RBA*RW*A2*A1*A0          ; read CAR
/WSAR1  = /RINGSEL + /RBA + RW + A2 + /A1 + /A0 ; write SAR
        ; RINGSEL*RBA*/RW*/A2*A1*A0 inverted, since active HIGH signal
WRAM1   = RINGSEL*RBA*/RW*A2*/A1*A0        ; write RAM during IO only
RRAM1   = RINGSEL*RBA*RW*A2*/A1*A0         ; read RAM normally
        + /IOMODE                          ; or in problem mode
```

FUSE MAP - RING BOARD PAL1

```
(C) Copyright 1987 MicroWay, Inc. All Rights Reserved.
PAL16L8
ALLAN STEPHENS  8/MARCH/88
CONTROL SIGNALS FOR LEAST SIGNIFICANT HALF OF RING

*D2217*F0*
L0000 1111 1111 1111 1111 1111 1111 1111 1111 *
L0032 1111 1111 0111 1111 1111 1111 1111 1111 *
L0256 1111 1111 1111 1111 1111 1111 1111 1111 *
L0288 1010 1111 1111 0111 1011 0111 1111 1111 *
L0512 1111 1111 1111 1111 1111 1111 1111 1111 *
L0544 1010 0111 1111 0111 0111 1011 1111 1111 *
L0768 1111 1111 1111 1111 1111 1111 1111 1111 *
L0800 1010 0111 1111 0111 0111 0111 1111 1111 *
L1024 1111 1111 1111 1111 1111 1111 1111 1111 *
L1056 1101 1111 1111 1111 1111 1111 1111 1111 *
L1088 0111 1111 1111 1111 1111 1111 1111 1111 *
L1120 1111 0111 1111 1111 1111 1111 1111 1111 *
L1152 1111 1111 1111 1111 1111 0111 1111 1111 *
L1184 1111 1111 1111 1111 1011 1111 1111 1111 *
L1216 1111 1111 1111 1011 1111 1111 1111 1111 *
L1536 1111 1111 1111 1111 1111 1111 1111 1111 *
L1568 1010 1011 1111 0111 1011 0111 1111 1111 *
L1792 1111 1111 1111 1111 1111 1111 1111 1111 *
L1824 1010 0111 1111 0111 1011 0111 1111 1111 *
L1856 1111 1111 0111 1111 1111 1111 1111 1111 *
ADF7
```

Appendix C : SIVMON Guide

This appendix describes the operation of the SIVMON 3.8 command interpreter.

SIVMON commands can be issued to the sieve either by a host computer running the OASiS software or by a human user sitting at a dumb terminal. This document uses the terms *host* and *user*, respectively, to designate these two sources, but since commands are processed in the same way in both cases, the terms are interchangeable. The use of a particular term in the following documentation normally denotes the source most likely to encounter the circumstances under discussion.

C.1 Command Basics

SIVMON prompts for a command by sending the greater than ('>') character. In problem mode only, this is followed by an exclamation mark ('!') if sieving has been halted by the detection of solutions and/or the trial counter reaching its maximum count. The '!' appears as soon as either of these conditions occurs, but will never interrupt a command after the first character has been typed. This notification is repeated on subsequent prompts as long as the halting condition(s) remain; the user should normally take this as a sign to halt sieving and investigate the cause of the stoppage.

Commands are entered by typing a single character mnemonic and zero or more additional command arguments or subcommmand mnemonics; extra blanks or other whitespace characters are not permitted anywhere. Alphabetic input must be given in upper case and numeric input in hexadecimal. Each command is interpreted as it is entered; once the required number of characters have been supplied, SIVMON automatically prompts for the next command. The user should *not* enter a carriage return at the end of a command. For this reason, incorrect input cannot be corrected by backspacing and typing over the mistake. If the user enters a command that is not permitted in the current mode, or enters an invalid subcommand or numeric argument, SIVMON immediately terminates the command and issues an appropirate error message.

The user can abort a command at any point by typing CTL-C. However, because characters are interpreted as they are typed, aborting a WRITE command in mid-execution

usually results in at least partial modification of the sieve component being written to. This also applies to the READ RING ALL subcommand, which alters the ring's CAR as it reads the ring RAM containing a congruence's residues.

C.2 INPUT/OUTPUT CONTROL

The input characters transmitted by the host are stored in an eight character input buffer as they are received and are removed as needed by SIVMON's command interpreter. Most input characters are echoed when they are taken from the buffer; however, the CANCEL (CTL-C), XOFF (CTL-S), and XON (CTL-Q) characters are not echoed at all.

The host can suspend the output produced by SIVMON by entering an XOFF character and resume it later by entering an XON.[1] This feature is most useful for commands that produce a large amount of output (eg. READ RING ALL). Entering XOFF while output is already suspended has no effect; nor does entering XON while output is not suspended. Under normal circumstances, the host should not enter input while output is suspended; any characters received (other than XON and XOFF) are simply buffered and are not echoed or interpreted until the suspension is lifted. If the buffer overflows because of continued typing by the host during a period of suspension, all of the characters within the buffer are lost.

Since the OAS acts as a dedicated slave to its host, SIVMON is expected to be able to keep up with the input transmitted to it. Thus, even though SIVMON reacts to XON and XOFF signals from its host, it never generates them itself. If data is transmitted to the sieve more quickly than they can be processed, input characters will eventually be lost. It is up to the host to prevent this situation from occurring; the OASiS software does this by waiting for each input character to be echoed by SIVMON before transmitting the next.

The host can abort a command by transmitting a CTL-C character. When SIVMON receives the character, it immediately flushes its input buffer and terminates the active command with the message 'COMMAND CANCELLED'. The command interpreter does not recognize the CTL-C character as a cancellation signal until an I/O operation is

---

1 This suspension has no effect on sieving, which continues even when command processing is blocked. This applies to both the search for and (if counting is enabled) the counting of solutions.

performed, so if CTL-C is entered while output is suspended, the active command is not aborted until a CTL-Q is typed.

C.3 COMMAND SUMMARY

Tables C.1 through C.3 list the commands available in each of the three sieve modes.

Table C.1
Commands Available in Sleeping Mode

Name	Mnemonic	Description
INITIALIZE	I	Reset sieve
QUERY	Q	Return sieve status
UP	U	Swtich to idle mode

Table C.2
Commands Available in Idle Mode

Name	Mnemonic	Description
COUNT	C	Enable solution counting mode
DOWN	D	Switch to sleeping mode
FIND	F	Enable solution recording mode
GO	G	Switch to problem mode & start sieving
INITIALIZE	I	Reset sieve
MEMORY	M	Examine M68701 memory location
QUERY	Q	Return sieve status
READ	RC	Read clock speed
	RM	Read solution mask
	RR	Read ring (has several options)
	RS	Read solution counter
	RT	Read trial counter
	RW	Read solution window
STEP	S	Sieve in problem mode for 1 cycle
WRITE	WC	Load clock speed
	WM	Load solution mask
	WR	Load ring (has several options)
	WS	Load solution counter
	WT	Load trial counter
	WW	Clear solution window

Table C.3
Commands Available in Problem Mode

Name	Mnemonic	Description
HALT	H	Stop sieving & switch to idle mode
INITIALIZE	I	Reset sieve
MEMORY	M	Examine M68701 memory location
QUERY	Q	Return sieve status

C.4 COMMAND DESCRIPTIONS

This section contains a brief description of each command provided by SIVMON, arranged alphabetically by mnemonic for easy reference. Each description includes:

- the syntax of the command,
- the sieve mode(s) in which the command is available, and
- the function(s) that the command carries out.

The functional description may also give guidance on how the command should be used, special situations to watch out for, and so on.

The designation <h*n*> in the syntax of a command means that exactly *n* hexadecimal digits are required at this point. The digits should *not* be enclosed with the '<' and '>' characters. For example, if the syntax of a command is 'X<h2>Y<h4>', then one possible invocation of this command would be 'X12Y0A7F'.

C.4.1 Count

Syntax: C

Modes: idle mode

Operation: Switches the sieve's search mode to solution counting.

Comments: If the sieve is already in this mode, the command has no effect.

C.4.2 Down

Syntax: D

Modes: idle mode

Operation: Puts the sieve into sleeping mode.

Comments: This command puts the sieve's rings under the control of an adjacent control board and should only be done if the other sieve unit is prepared to take charge of them.

C.4.3 Find

Syntax: F

Modes: idle mode

Operation: Switches the sieve's search mode to solution recording.

Comments: If the sieve is already in this mode, the command has no effect.

C.4.4 Go

Syntax: G

Modes: idle mode

Operation: Clears the solution bus of any solutions, then runs the sieve in problem mode until HALTed.

Comments: If the trial counter is currently in its maximum count state, the sieve clock is immediately blocked and no clock pulses are generated.

C.4.5 Halt

Syntax: H

Modes: problem mode

Operation: Stops the sieve clock, then returns the sieve to idle mode.

Comments: The clock may already be blocked by the trial counter reaching its maximum count state, or the appearance of solutions on the solution bus, or both.

C.4.6 Initialize

Syntax: I

Modes: idle mode, problem mode, or sleeping mode (ie. valid anytime)

Operation: Does the software equivalent of a hardware RESET.

Puts the sieve in idle mode, with its search mode set to solution recording. Sets the serial communication port speed to 9600 baud, and format to 8 bit, no parity. Issues a sign-on message of 'SIVMON 3.8' and prompts the user for a command with '>'.

The sieve clock is set to use the fastest possible sieving speed. No other sieve components are altered.

C.4.7 Memory

Syntax: M<h4>

Modes: idle mode, problem mode

Operation: Displays the contents of memory at the specified address within the M68701's address space, then waits for a user response. To alter the value of the specified byte, enter '=' then the two hexadecimal digits specifying the new value; to leave the byte unchanged, simply enter a carriage return.

Comments: This command makes no attempt to verify that any replacement value is successfully written; thus it will not complain if the user writes to EPROM or non-existent memory locations.

This command should never be used under normal circumstances, but may prove useful when debugging the sieve hardware (or SIVMON itself).

C.4.8 Query

Syntax: Q

Modes: idle mode, problem mode, or sleeping mode (ie. valid anytime)

Operation: Displays a four character string which describes the current status of sieve operations.

The first character gives the current *sieve mode*, where:

X : sleeping mode
I : idle mode
P : problem mode

The second character gives the current *search mode*, where:

R : solution recording (ie. stop when solution is found)
C : solution counting

The third character gives the current *solution bus status*, where:

S : solution(s) exist on solution bus
- : no solutions exist

The final character gives the current *problem done status*, where:

D : problem done (trial counter has reached maximum count)
- : problem not done

Comments: If the sieve mode is 'X' (sleeping), the remaining characters returned by this command are blanks, indicating that the other pieces of status information are not available.

C.4.9 Read

General: All of the sieve devices used during the execution of a sieving problem can be read using one of the following subcommands of READ. Each of these subcommands can be performed only in idle mode.

C.4.9.1 READ CLOCK SPEED

Syntax: RC

Modes: idle mode

Operation: Displays the current sieve clock "divide by" value as a single hex digit.

Comments: See the description of the WRITE CLOCK SPEED subcommand for a table of the available clock speeds.

C.4.9.2 READ SOLUTION MASK

Syntax: RM

Modes: idle mode

Operation: Displays the 16 bits of the solution mask as four hex digits.

Comments: A zero bit indicates that the corresponding solution bus line is currently disabled (permanently zero).

C.4.9.3 READ RING

General: All subcommands of this command begin with the mnemonic 'RR<h2>', where the two hex digits specified must denote a ring number in the range 00_{16} to $1F_{16}$ (0 to 31 decimal). Although SIVMON generates an error message if the value is not in this range, it does *not* check to see if the current hardware configuration actually contains a ring with that number.

C.4.9.3.1 Read Ring All

Syntax: RR<h2>A

Modes: idle mode

Operation: Displays the specified ring's 16-bit SAR and CAR values as 4 hex digits each; then displays the contents of all RAM locations used (addresses SAR to $FFFF_{16}$, inclusive), also as 4 hex digits each.

C.4.9.3.2 Read Ring CAR

Syntax: RR<h2>C

Modes: idle mode

Operation: Displays the 16 bits of the specified ring's Current Address Register as 4 hex digits.

C.4.9.3.3 Read Ring RAM

Syntax: RR<h2>R

Modes: idle mode

Operation: Displays the 16 bits of the specified ring's Random Access Memory (whose address corresponds to the ring's current CAR value) as 4 hex digits.

C.4.9.3.4 Read Ring SAR

Syntax: RR<h2>S

Modes: idle mode

Operation: Displays the 16 bits of the specified ring's Start Address Register as 4 hex digits.

C.4.9.4 READ SOLUTION COUNTER

Syntax: RS

Modes: idle mode

Operation: Displays the 64 bits of the solution counter as 16 hex digits.

C.4.9.5 READ TRIAL COUNTER

Syntax: RT

Modes: idle mode

Operation: Displays the 48 bits of the trial counter as 12 hex digits.

C.4.9.6 READ SOLUTIONWINDOW

Syntax: RW

Modes: idle mode

Operation: Displays the 16 bits of the solution window as 4 hex digits.

Comments: A "1" bit indicates a solution has been found on the corresponding line of the solution bus.

C.4.10 Step

Syntax: S

Modes: idle mode

Operation: Clears the solution bus of any solutions, then runs the sieve in problem mode for one clock cycle before returning it to idle mode.

Comments: If the trial counter is currently in its maximum count state, the sieve clock is immediately blocked and no clock pulses are generated.

C.4.11 Up

Syntax: U

Modes: sleeping mode

Operation: Returns sieve to idle mode.

Comments: All sieve components (with the possible exception of the rings) should be as they were when the sieve was put to sleep. The rings are changed only if another sieve control board has used them in the interim (either by writing to the rings or by running the sieve in problem mode for one or more cycles).

C.4.12 Write

General: All of the sieve devices used during the execution of a sieving problem can be written using one of the following subcommands of WRITE, with the exception of the solution window which can only be cleared. Each of these subcommands can be performed only in idle mode.

C.4.12.1 WRITE CLOCK SPEED

Syntax: WC<h1>

Modes: idle mode

Operation: Sets the current sieve clock "divide by" value to the hex digit specified.

Table C.4
Sieving Speeds for a 40 MHz Crystal

divide by	clock (MHz)	trials per second (16 taps)
3	13.3	213 333 333
4	10	160 000 000
5	8	128 000 000
6	6.6	106 666 666
7	5.7	91 428 571
8	5	80 000 000
9	4.4	71 111 111
10	4	64 000 000

Comments: It is an error to specify a value outside the range 3 to 10 (decimal).

C.4.12.2 WRITE SOLUTION MASK

Syntax: WM<h4>

Modes: idle mode

Operation: Loads the 16 bits of the solution mask with the four hex digits specified.

Comments: A zero bit indicates that the corresponding solution bus line is to be disabled (permanently zero).

C.4.12.3 WRITE RING

General: All subcommands of this command begin with the mnemonic 'WR<h2>', where the two hex digits specified must denote a ring number in the range 00_{16} to $1F_{16}$ (0 to 31 decimal). Although SIVMON generates an error message if the value is not in this range, it does *not* check to see if the current hardware configuration actually contains a ring with that number.

C.4.12.3.1 Write Ring All

Syntax: WR<h2>A<h4><h4><h*n*>

Modes: idle mode

Operation: Loads the specified ring's 16-bit SAR and CAR with the first two 4-digit hex values supplied; then loads the contents of all RAM locations used (addresses SAR to $FFFF_{16}$, inclusive) with the remaining 4-digit hex values supplied. Thus, $n = 4 \times (10000_{16}$ - SAR).

Comments: SIVMON does *not* check that the CAR value supplied is greater than or equal to the SAR value supplied, as it must be in order for the ring to function properly.

C.4.12.3.2 Write Ring CAR

Syntax: WR<h2>C<h4>

Modes: idle mode

Operation: Loads the 16 bits of the specified ring's Current Address Register with the four digit hex value supplied.

Comments: SIVMON does *not* check that the CAR value supplied is greater than or equal to the ring's current SAR value, as it must be in order for the ring to function properly.

C.4.12.3.3 Write Ring Load

Syntax: WR<h2>L<h6><h*n*>

Modes: idle mode

Operation: Loads the specified ring with the congruence supplied. The congruence's modulus is given as the first 6 hex digits. The remaining hex digits correspond to the residues of the congruence, padded to a multiple of 16 bits (4 hex digits) if necessary. Thus, $n = 4 \times \lceil \text{modulus} \div 16 \rceil$. A "1" bit corresponds to an acceptable residue.

Comments: When sieving begins, the ring presents the first 16 residues of the congruence, then the next 16, and so on, in a cyclic fashion.

SIVMON does *not* check for a modulus value that is too large to be supported by the specified ring's hardware, or for modulus value of zero. A congruence of modulus m will fit in a ring containing k RAM locations if $\text{lcm}(m,16) \leq (k \times 16)$.

This command is the best method of loading a ring with the residues for a congruence, since it automatically generates enough copies of the residues to fill an integral number of RAM locations. The replication of residues takes virtually no time in comparison with the time required to transmit them, so WRITE RING LOAD can load a given congruence up to 16 times faster than the corresponding WRITE RING ALL command.

C.4.12.3.4 Write Ring RAM

Syntax: WR<h2>R<h4>

Modes: idle mode

Operation: Loads the 16 bits of the specified ring's Random Access Memory (whose address corresponds to the ring's current CAR value) with the four digit hex value supplied.

C.4.12.3.5 Write Ring SAR

Syntax: WR<h2>S<h4>

Modes: idle mode

Operation: Loads the 16 bits of the specified ring's Start Address Register with the four digit hex value supplied.

Comments: SIVMON does *not* check that the SAR value supplied is less than or equal to the ring's current CAR value, as it must be in order for the ring to function properly.

C.4.12.3.6 Write Ring Verify

Syntax: WR<h2>V<h6><h*n*>

Modes: idle mode

Operation: Verifies the specified ring with the congruence supplied. The congruence's modulus is given as the first 6 hex digits. The remaining hex digits correspond to the residues of the congruence, padded to a multiple of 16 bits (4 hex digits) if necessary. Thus, $n = 4 \times \lceil \text{modulus} \div 16 \rceil$. A "1" bit corresponds to an acceptable residue.

After the modulus has been entered, SIVMON returns P(ass) if the corresponding SAR value used for such a congruence agrees with the ring's actual SAR value; if they disagree, SIVMON returns F(ail) and terminates the command.

Once the modulus has been verified, SIVMON returns P(ass) after each 4 hex digit segment of residues has been verified; if any of the copies of a residue segment does not match the ring's actual RAM contents, SIVMON returns F(ail) and terminates the command.

Comments: Verification begins by comparing the first 16 residues of the congruence against the RAM contents at the current CAR (and any copies), then the next 16 against the RAM contents at CAR+1, and so on.

SIVMON does *not* check for a modulus value that is too large to be supported by the specified ring's hardware, or for modulus value of zero. A congruence of modulus m will fit in a ring containing k RAM locations if $\text{lcm}(m,16) \le (k \times 16)$.

This command is the best method of verifying the contents of a ring loaded using the WRITE RING LOAD command, since it automatically tests all copies of the residues. However, if the second phase of verification fails, the READ RING ALL command will be required to determine whether the ring's CAR or RAM is actually at fault.

C.4.12.4 WRITE SOLUTION COUNTER

Syntax: WS<h16>

Modes: idle mode

Operation: Loads the 64 bits of the solution counter with the 16 hex digits specified.

C.4.12.5 WRITE TRIAL COUNTER

Syntax: WT<h12>

Modes: idle mode

Operation: Loads the 48 bits of the trial counter with the 12 hex digits specified.

Comments: The trial counter should be loaded with the 1's complement of the maximum number of clock cycles to be performed. For example, if at most one cycle of the sieve clock is be performed, the trial counter should be loaded with $FFFFFFFFFFFE_{16}$.

C.4.12.6 WRITE SOLUTION WINDOW

Syntax: WW

Modes: idle mode

Operation: Sets the 16 bits of the solution window to all zeroes, thus removing any solutions from the solution bus.

Comments: Since the sieve clock is blocked by the presence of one or more solutions on the solution bus, this function must be performed before sieving can be resumed. However, the GO and STEP commands do this automatically before starting the sieve clock, so it is unnecessary for the user to issue this command under normal circumstances.

Appendix D : SEND Guide

This appendix describes the OASiS SEND command, which can be used as an aid in debugging the Open Architecture Sieve hardware.

D.1 THE OASIS SEND FACILITY

Testing the Open Architecture Sieve by having OASiS run sample problems is not always the best way of diagnosing an apparent sieve failure for two reasons: it involves considerable overhead (resulting in slow turnaround) and it requires using OASiS software which may itself contain errors. Instead, the user is often better off connecting a terminal to the sieve's control board and entering a sequence of SIVMON commands by hand. This allows the user to control the sieve hardware in the same way OASiS does when running a problem, but without being restricted to the standard sequence of commands that OASiS uses in solving a sieving problem. Thus, if the user understands how the sieve is supposed to operate, one or more carefully chosen commands can be used to determine if the hardware is functioning correctly and, if not, their responses used to guide further command sequences that attempt to narrow down the exact nature of the failure.

Using this technique to diagnose certain sorts of errors is relatively easy—for example, the questions "Is the sieve responding at all?" or "Is the trial counter counting?" can be answered by issuing a handful of simple commands. However, testing other sources of error may require the user to send long sequences of commands that initialize the sieve hardware in a particular way prior to performing an actual test. Entering these commands by hand is a very slow and error-prone process. If a non-trivial sequence must be performed several times—for example, to allow the sieve's innards to be probed with an oscilloscope or logic analyzer, or to judge the effects of repairs—debugging can be slowed to a negligible pace. For this reason, the OASiS SEND facility has been developed to allow command sequences to be transmitted with a minimum of effort.

The SEND command allows a user to create a file containing a pre-written *script* of SIVMON commands and have it transmitted to the sieve, or to develop and transmit such scripts interactively. A pre-written script is most useful when a test is long or must be

repeated, while an interactive script is best if the determination of what commands are to be issued is at least partially based on the results of earlier commands.

The scripts used by SEND look very much like lines of SIVMON commands, and can be easily developed using an ordinary text editor. SEND displays the commands it transmits to the sieve and the responses that the sieve generates (when told to look for them). It can compare the sieve's responses with the responses expected by the user and generate an error message if the two do not agree. It also notifies the user of any communication errors with the sieve.

The SEND facility can be an invaluable tool in debugging the sieve hardware, but it does have some limitations which must be clearly understood at the outset.

1) SEND has no understanding of SIVMON beyond the communication protocol that it uses and prompt that it issues after each command. The user must have a clear understanding of what the SIVMON commands are, which commands generate responses, and what each response looks like. SEND blindly follows whatever instructions it is given, regardless of whether or not they make any sense to SIVMON.

2) The special sequences that tell SEND what to do are not checked for syntax errors. Using a script that contains sequences that SEND does not understand produces unpredictable results and may simply cause the command to "hang".

3) There is no mechanism for writing control structures (such as loops and if-then-else constructs) into scripts, so pre-written scripts must be strictly sequential in nature. A command's response cannot be used to influence the commands that follow it—a response can only be compared against a user-supplied response and any differences reported.

None of these restrictions is particularly major. Learning the SIVMON command set is not difficult, nor is ensuring that a script is composed correctly. If a mistake is made and the script is contained in a file, it is usually a simple matter to make the appropriate correction and retransmit the file. Admittedly, allowing a script to have control mechanisms and "local variables" would be nice, but the lack of such features has not proved to be a major drawback during the author's own debugging experiences. For now, the user can get non-sequential processing by using an interactive script and making the control decisions manually.

D.2 USING THE SEND COMMAND

This section describes how the SEND command can be used. For the best results, a user should read this section carefully before using SEND to learn exactly what features it provides and get hints on how they can be used most effectively.

D.2.1 Before Using SEND

To use the SEND command, the user first needs to sign on to the MicroVax which acts as host to the OAS. It it often helpful if the user's terminal is adjacent to the sieve to allow its responses to be observed directly or to allow adjustments to be made to its hardware. The following steps should then be performed.

1) Sign on to the MicroVax as username OASIS.
2) If necessary, configure the sieve's terminal line to run at 9600 baud using 8 bit, no parity communication (/FRAME=8, /PARITY=NONE). The /HOSTSYNC, /NOWRAP, and /NOBROADCAST options should be enabled so the MicroVax will use the XON/XOFF protocol, will not send an extra linefeed to the sieve every 80 characters, and not pass broadcast messages generated by the system administrator from interfering with the operation of the sieve.
3) If the sieve is not powered up, disconnect it from its terminal line, then power it on; after waiting a few seconds, reconnect the line. Failure to do this may result in SIVMON's sign-on message being interpreted by the MicroVax as an attempt by the sieve to sign on as an interactive user.

D.2.2 Invoking SEND

The SEND facility is invoked by the following command.

$ send [*filename*] [*portname*]

The command transmits the contents of the file *filename* to the OAS over the terminal line *portname*. A file extension of "TXT" is assumed for *filename* if no extension is specified. If *filename* is omitted, SEND takes its input from SYS$INPUT (which defaults to the user's terminal). If *portname* is omitted, SEND uses the default terminal line "TXF2:". If either the file or the terminal line cannot be opened, an error message is issued. To indicate that *filename* is missing when *portname* is present, the null string ("") should be entered for *filename*.

As far a SEND is concerned, it is irrelevant whether the script being processed is contained in a file or is being typed in by the user. However, the user should be aware of several points concerning interactive scripts that may otherwise cause confusion.

1) SEND does not prompt for the first line of user input—simply begin typing once it has been invoked.
2) Each SIVMON command is terminated by hitting the return key; the command is not read by SEND until the return, so any typing mistakes can be corrected by backspacing over the error and retyping.
3) SEND displays the characters of a SIVMON command as they are transmitted to the sieve, so each command is redisplayed on the line following the user's input. (With a pre-written script each command appears only once.)
4) End-of-file is indicated by entering CTL-Z (without hitting the return key.)

D.2.3 Basic Script Techniques

The most basic form of a script consists of SIVMON commands, one to a line.

D.2.3.1 Sending Normal Characters

SEND transmits most ASCII characters to the sieve as they are read in and then displays the character. If a transmission error occurs, it prints a message specifying the cause of the error. When SEND reaches the end of a line, it assumes that the SIVMON command is finished and looks for the carriage return, linefeed, '>' prompt that should follow.

Thus, a simple script which writes to the trial counter and then starts sieving might look like

```
WT123456789012<nl>
G<nl>
```

where "<nl>" indicates the end of each line. When used as input by SEND, the output displayed on the terminal would be

```
WT123456789012
>G
>
```

with the prompt characters '>' being supplied by the sieve. SEND requires that the script follow SIVMON's syntax rules carefully. No extra spaces are permitted, all characters must be in upper case, and numbers must be given in hexadecimal form.

D.2.3.2 READING SIEVE RESPONSES

The *read* sequence '?n.' in a comand causes SEND to read *n* characters from the sieve and display them. The number of characters to be read is given in decimal (not hex) and must be in the range 0 to 256. The period after the number is necessary to delimit the number from any remaining characters of the command.

For example, to read the twelve digit trial counter, the command

```
RT?12.<nl>
```

could be used. This would transmit 'RT' to the sieve, display the next 12 characters sent back, and then look for SIVMON's command prompt.

D.2.3.3 VERIFYING SIEVE RESPONSES

The *assert* sequence '!n.' is like '?n.' except that SEND compares the characters read with the next *n* characters on the command line. If the sieve's response is correct, SEND does nothing else (ie. '!n.' acts just like '?n.'). If they do not match, it issues a message and prints the two strings on adjacent lines to allow the user to inspect their differences.

For example, the trial counter could be checked to see if it had the value 1 by sending the following command.

```
RT!12.000000000001<nl>
```

D.2.4 Advanced Script Techniques

With only the transmit, read, and assert functions, it is possible to write scripts that can perform most operations required during debugging. However, some situations are more easily handled using more sophisticated features.

D.2.4.1 THE PAUSE SEQUENCE

It is sometimes necessary to have the sieve run in problem mode for a certain amount of time before checking its results. The *pause* sequence '#n.' causes SEND to wait for roughly *n* seconds before continuing the processing of a script. The value of *n* can be any positive integer.

Thus, initializing the sieve with a problem and then sending the sequence

```
G<nl>
#10.H<nl>
```

runs the problem for 10 seconds. Note that it is incorrect to code

```
G<nl>
#10.<nl>
H<nl>
```

since doing so would cause SEND to look for SIVMON's command prompt when it resumed from its pause, which would not have been generated. Remember that "pause" is an instruction to SEND, not to SIVMON.

D.2.4.2 SENDING SPECIAL CHARACTERS

To transmit unprintable characters (such as control characters), or characters that have a special meaning to SEND (such as '?', '!', '#', '*', '\', ';', and <nl>), the user must specify '\n.' where *n* is the decimal value for the ASCII character.

For example, to initialize the OAS and swallow the initialization message that follows (ie. carriage return, linefeed, '`SIVMON 3.8`'), the following command can be specified.

```
I!12.\13.\10.SIVMON 3.8<nl>
```

Since the only other SIVMON command that uses a non-alphanumeric is MEMORY (which uses a carriage return to indicate "no change"), this feature is seldom necessary except in initializing the sieve.

SEND knows enough about the SIVMON communication protocol so that it does not expect the special characters CANCEL (CTL-C), XOFF (CTL-S), and XON (CTL-Q) to be echoed by SIVMON.

D.2.4.3 THE REPETITION SEQUENCE

When writing or reading sieve components it is often necessary to specify the same character a number of times in succession. This can be tedious if the number of repetitions is large, so SEND allows the user to indicate *n* occurrences of a character by prefixing it with the *repetition* sequence '*n.'. The number of repetitions can be any non-negative number, but the zero repetition factor simply consumes the following character and transmits nothing.

For example, the following command writes all zeroes to the trial counter.

```
WT*12.0<nl>
```

When this is transmitted by SEND , the repetition sequence is displayed as twelve consecutive zeroes.

The repetition sequence *cannot* be used to repeat an entire command. For example,

```
*100.S<nl>
```

would not generate 100 STEP commands, since a single STEP command consists of the following *two* character sequence.

```
S<nl>
```

The first version would simply generate 100 consecutive S characters with no intervening newline character. (Future versions of SEND may support repetition of sequences of characters.)

D.2.4.4 COMMENTING SCRIPTS

Scripts can be commented for the benefit of other sieve users. SEND ignores all characters on a line from a semicolon (';') up to and including the end of the line.

The best way of using the comment facility is to put each comment on a line by itself. For example, the user can document an instruction or instruction sequence by doing the following.

```
; read trial counter<nl>
RT?12.<nl>
```

In most cases, a comment should *not* go on the same line as a SIVMON command. For example,

```
RT?12.          ; read trial counter<nl>
```

would cause problems because the whitespace character(s) before the comment would be transmitted to SIVMON. Even

```
RT?12.;         read trial counter<nl>
```

would not work because the end-of-line character after the comment would be ignored and SEND would not look for the prompt following the end of the command. The sequence

```
RT?12.;         read trial counter<nl>
<nl>
```

would work, but this would lead to problems if the comment was later removed without removing the extra newline character.

Another side effect of the comment feature definition allows it to be used to spread a long SIVMON command over a number of lines. For example, the trial counter could be loaded in the following manner.

```
WT000000;<nl>
000000<nl>
```

The command works because the comment causes the newline character at the end of the line to be ignored. The command splitting facility is probably most useful when writing or verifying a ring containing a large number of residue classes. Just be sure to omit the trailing semicolon from the last line of the command!

D.3 MODIFYING THE SEND FACILITY

It should not be necessary to modify the SEND facility unless the user wishes to add to its capabilities or to fix bugs. However, the following directions are given should this situation arise.

The SEND command is implemented by a program written in C and located in the file [OASIS.SOFTWARE.SCRIPT]SEND.C. The code is relatively straightforward and contains numerous comments explaining its operation. The mainline consists of a simple loop that processes characters from the script file, either transmitting a single character, processing a read or assert sequence, or swallowing the prompt that follows a command; after each action, it checks for errors and generates a message if one has occurred. The characters read in by the mainline are obtained by calling a subroutine that takes care of:

- translating the read, assert, and newline characters to the internal form used by the mainline,
- translating unprintable characters into internal ASCII form,
- implementing the repetition sequence, and
- implementing the wait sequence.

The subroutine also strips any comments from the script, so they do have to be handled by the mainline. The mainline communicates with the OAS by calling the same interface routines that are used by the SIEVE command.

After the necessary changes have been made, the OASiS software makefile located in the directory [OASIS.SOFTWARE] can be used to recompile the SEND program and link in the necessary library routines. The modified program will be automatically invoked the next time the SEND command is issued.

Appendix E : The OASiS Problem File

An OASiS problem file is an ordinary ASCII text file containing a description of the problem to be solved. The description is written in a simple language which allows the problem to be defined in a concise but natural manner, allowing the user great freedom in defining the problem while still permitting OASiS to detect many mistakes. Many of the language's restrictions correspond to limitations imposed by the way OASiS represents a problem internally during execution; for example, certain numeric values are limited to a specific number of digits to prevent overflow. Other restrictions allow OASiS to detect semantic errors, such as specifying a residue for a congruence which is larger than the modulus. The generous limits accompanying most restrictions mean that the user should seldom have difficulty creating a valid problem file.

E.1 Problem File Layout

The language used for defining problems allows a relatively free format layout similar to that used by a programming language like Pascal. A problem is made up of several different types of syntactic units or *tokens*: keywords, identifiers, quoted strings, and integers. These tokens are delimited by the first character that cannot be a part of the current token or by the special characters (blank, tab, and newline) known as *whitespace*. Whitespace may be entered freely between tokens to improve readability, but may not appear within a token; for example, the keyword 'FILTER' may not be written as 'FIL TER'. Alphabetic characters may be given in upper case, lower case, or a mixture of both, since OASiS converts everything to upper case internally.

User comments are specified by placing the remark after an exclamation mark ('!'). The comment is terminated by an exclamation mark, newline character, or end-of-file, whichever comes first. A comment may appear anywhere whitespace may appear.

E.2 Problem File Contents

A problem file has four parts: a *name part*, a *congruences part*, a *parameters part*, and a *results part*. The first three parts define the problem to be solved and are required parts of

every problem file. The results part contains messages generated during the execution of the problem and is normally omitted when the problem file is created. If the results part is present, OASiS resumes execution of the problem from the point at which it terminated rather than from the beginning. The four parts of a problem file must appear in the order given above.

E.2.1 Name Part

The name part has the form

PROBLEM *problem_name*

where *problem_name* can be any alphanumeric string from 1 to 32 characters in length. OASiS does not use the problem name for anything during its processing, but the name can be used as a means of identifying a particular problem file.

E.2.2 Congruences Part

The congruences part describes the set of congruences to be solved. It consists of the keyword 'CONGRUENCES' and one or more *congruence descriptions.*

E.2.2.1 CONGRUENCE DESCRIPTION

Each congruence description defines a single congruence. It has the form

$$modulus : residue_1\ residue_2 \ldots residue_n ;$$

where *modulus* denotes the modulus of the congruence and the remaining values list the acceptable residues for that congruence. The modulus value must be in the range 2 to 99999. The residue list consists of one or more integer values separated by whitespace, each of which must be in the range 0 to *modulus*-1. The colon separator (':') between the modulus and the first residue and the semicolon terminator at the end of the residue list (';') are required parts of the congruence description.

Residue values can be given in any order, although placing them in ascending order is usually most sensible. For convenience, a set of contiguous residues may be specified by placing a colon between two values, with the smaller value occurring first; this specifies all residues from the smaller value up to and including the larger value. It is an error to

specify a given residue more than once in a congruence, to specify all of the possible residues, or to specify no residues at all.

There is no limit on the number of congruence descriptions that a problem file may contain, nor on the order in which they appear. However, it is an error to specify the same modulus in more than one congruence.

E.2.2.2 CONGRUENCE GROUPS

Normally, the user is not concerned with how the congruences are to be loaded into the available rings; OASiS automatically attempts to find a mapping of congruences into rings that maximizes the number of congruences being processed in hardware, and thus maximizes the efficiency of the sieving process. Occasionally, however, a user may find it necessary to force the rings to be loaded in a certain manner to achieve an acceptable mapping. There are two methods provided for controlling how congruences are mapped into rings: one which specifies congruences that should be loaded into the hardware rings, and one which specifies congruences that should not be loaded.

A given set of congruences can be forced into a single hardware ring by forming a *congruence group*. This is done by enclosing one or more consecutive congruence descriptions in round brackets and prefixing the collection with the keyword 'RING'. OASIS then merges the congruences and assigns the resulting congruence to its own hardware ring, if possible; the congruences are never split up over several rings, nor are any other congruences put in the same ring with the group. Although congruence groups have the highest priority in OASiS' ring allocation algorithm, there is no guarantee that a given congruence group will always be placed into a hardware ring since the size and number of rings provided by the hardware is limited; if no ring can be found for the group, OASiS implements the group using a software ring.

In contrast, one or more congruences can be kept out of the hardware rings by forming a congruence group which is prefixed with the keyword 'NONRING' instead of 'RING'; OASiS always implements the specified congruences using software rings, even if some hardware rings go unused as a result.

There is no restriction on the number of congruences in a congruence group, nor on the number of congruence groups in a problem file.

E.2.3 Parameters Part

The parameter part describes how sieving is performed on the congruences specified in the congruences part. It consists of the keyword 'PARAMETERS' and a *search description*. Optional *solution limit*, *time limit*, *filter specification*, *error limit*, and *checkpoint interval* constructs may also follow, in that order.

E.2.3.1 Search Description

The search description has the form

how FROM *low_bound* [TO *high_bound*]

where *how* is one of 'RECORD' or 'COUNT', and both *low_bound* and *high_bound* are non-negative integers of any size. If present, *high_bound* must be larger than *low_bound*. The sieve searches the interval [*low_bound*,*high_bound*), either reporting each solution as it is found, or just keeping track of the number of solutions it finds, depending on the value of *how*.[1] The 'TO' portion may be omitted if no upper limit on the range of searching is desired.

E.2.3.2 Solution Limit

The solution limit instructs OASiS to halt sieving once the desired number of solutions has been found. It has the form

SOLUTION LIMIT = *number*

where *number* is a positive integer. The solution limit may be omitted if none is desired.

E.2.3.3 Time Limit

The time limit instructs OASiS to halt sieving once the specified amount of time has been spent on the problem. It has the form

TIME LIMIT = *time_amount*

where *time_amount* consists of an integer in the range 1 to 10^9-1 followed by one of: 'SECONDS', 'MINUTES', 'HOURS', or 'DAYS'. There is a limit of approximately 10^9 seconds (over 30 years) on sieve problems. This limit is cumulative; if a problem has a 4

---

1 Currently the COUNT mode is not supported by OASiS.

hour limit and is stopped after 3 hours, it runs for only one more hour once it is restarted and then terminaties automatically. The time limit may be omitted if none is desired.

E.2.3.4 FILTER SPECIFICATION

The filter specification names the file containing the filter program used by the problem and the maximum amount of time a call to the filter routine is allowed to run before an error is generated. It has the form

FILTER = *'file_name'*

FILTER LIMIT = *time_amount*

where *file_name* is a string of 1 to 63 characters enclosed in single quotes and *time_amount* is as described previously. This specification may be omitted if no filtering is to be done.

E.2.3.5 ERROR LIMIT

The error limit specifies the number of non-fatal errors that are permitted before problem execution is abandoned. It has the form

ERROR LIMIT = *number*

where *number* is in the range 1 to 9. The default error limit is 3, meaning the third non-fatal error terminates the problem. A problem's error count is reset to zero each time the problem resumes execution; thus, errors from a previous run are not counted in subsequent runs.

A non-fatal error occurs whenever OASiS gets a "bad" response from the sieve hardware, indicating that a malfunction has occurred. Any error involving the host mini-computer itself is considered fatal, and immediately terminates the execution of the problem regardless of the error limit specified.

E.2.3.6 CHECKPOINT INTERVAL

The checkpoint interval specifies how often OASiS lets the sieve hardware run before verifying that its contents are correct. The specification has the form

BACKUP EVERY *time_amount*

where *time_amount* is as described previously. The default interval between checkpoints is one hour.

E.2.4 Results Part

The results part consists of the keyword 'LOG' and a list of zero or more messages specifying the results of executing the problem. Each message is prefixed with the time at which it was issued; the time is given in the form 'DDDDD HH:MM:SS' and specifies the total time that the problem has been executing rather than local time. The messages themselves fall into four classes: informatory ('MESSAGE'), solution ('SOLUTION'), checkpoint ('VERIFIED'), and error ('ERROR'), and are usually self-explanatory. Appendix G, "A Sample Problem", contains an example of the messages generated by running a problem. As this part of the problem file is not normally created by the user, no further description of its form will be given.

E.3 PROBLEM FILE GRAMMAR

The following two pages contain a Backus-Naur form description of the OASiS problem file syntax. Non-terminal symbols are enclosed in '< >', while terminal symbols are character strings which are not enclosed in '< >'. Alternatives are separated by '|'. Terminals may be entered in either upper or lower case.

"Whitespace" (blanks, tabs, and newlines) may occur between the terminals and non-terminals making up a non-terminal, except in those non-terminals denoted with '†'. Whitespace is not permitted within a terminal. User-supplied comments begin with '!' and run up to the next '!', newline, or end-of-file. Comments may appear anywhere that whitespace may appear.

```
<problem file>          ::=  <name part> <congruences part> <parameter part>
                             <log part>

<name part>             ::=  PROBLEM <identifier32>

<congruences part>      ::=  CONGRUENCES <congruences>
<congruences>           ::=  <congruence type> <congruences> | <congruence type>
<congruence type>       ::=  <congruence group> | <congruence>
<congruence group>      ::=  <group type> ( <congruence list> )
<group type>            ::=  RING | NONRING
<congruence list>       ::=  <congruence> <congruence list> | <congruence>

<congruence>            ::=  <modulus> : <residue list> ;
<modulus>               ::=  <integer5>
<residue list>          ::=  <residue range> <residue list> | <residue range>
<residue range>         ::=  <residue> : <residue> | <residue>
<residue>               ::=  <integer5>

<parameter part>        ::=  PARAMETERS <required parms> <optional parms>
<required parms>        ::=  <search description>
<optional parms>        ::=  <solution limit> <time limit> <filter specification>
                             <error limit> <backup time>

<search description>    ::=  <search start> <search end>
<search start>          ::=  <search type> FROM <big integer>
<search type>           ::=  RECORD | COUNT
<search end>            ::=  TO <big integer> | <empty>

<solution limit>        ::=  SOLUTION LIMIT = <big integer> | <empty>

<time limit>            ::=  TIME LIMIT = <time amount> | <empty>

<filter specification>  ::=  <filter file> <filter time limit> | <empty>
<filter file>           ::=  FILTER = <file name>
<filter time limit>     ::=  FILTER LIMIT = <time amount>
<file name>             ::=  <quoted string63>

<error limit>           ::=  ERROR LIMIT = <integer1> | <empty>

<backup time>           ::=  BACKUP EVERY <time amount> | <empty>

<log part>              ::=  LOG <log line list> | <empty>
<log line list>         ::=  <log line> <log line list> | <log line> | <empty>
<log line>              ::=  <time> <log msg>
<log msg>               ::=  <message msg> | <error msg> | <solution msg> |
                             <verification msg>

<message msg>           ::=  MESSAGE <toendofline>
<error msg>             ::=  ERROR <toendofline>
<solution msg>          ::=  SOLUTION <solution number> AT <solution>
<verification msg>      ::=  VERIFIED <solution number> SOLUTIONS TO
                             <checkpoint>
```

<solution number>	::=	<big integer>
<solution>	::=	<big integer>
<checkpoint>	::=	<big integer>
<time amount>	::=	<integer$_9$> <time units>
<time units>	::=	HOURS | MINUTES | SECONDS | DAYS
<time>	::=	<days> <hours> : <minutes> : <seconds>
<days>	::=	<integer$_5$>
<hours>	::=	<integer$_2$>
<minutes>	::=	<integer$_2$>
<seconds>	::=	<integer$_2$>
<big integer>†	::=	<integer>E<integer$_2$> | <integer>
<integer>†	::=	<digit> <integer> | <digit>
<integer$_n$>†	::=	<digit> <integer$_{n-1}$> | <digit>
<integer$_0$>	::=	<empty>
<identifier$_n$>†	::=	<alphanumeric> <identifier$_{n-1}$> | <alphanumeric>
<identifier$_0$>	::=	<empty>
<alphanumeric>	::=	<alphabetic> | <digit>
<alphabetic>	::=	A | B | ... | Z | a | b | ... | z
<digit>	::=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<quoted string$_n$>†	::=	'<non-quote list$_n$>'
<non-quote list$_n$>†	::=	<non-quote> <non-quote list$_{n-1}$> | <non-quote> | <empty>
<non-quote list$_0$>	::=	<empty>
<non-quote>	::=	any ASCII character except '
<toendofline>†	::=	<non-newline list> <newline> | <newline>
<non-newline list>†	::=	<non-newline> <non-newline list> | <non-newline>
<non-newline>	::=	any ASCII character except a newline
<newline>	::=	ASCII newline character
<empty>	::=	

Appendix F : The OASiS Configuration File

An OASiS configuration file is an ordinary ASCII text file containing a description of the Open Architecture Sieve hardware resources available when a problem is executed. The description is written in a simple language which allows these resources to be specified as succinctly as possible and yet be easily understood by the user.

F.1 CONFIGURATION FILE LAYOUT

The language used for defining a configuration is based on the same principles as that used for defining problems (see the section "Problem File Layout" in Appendix E). Thus, the configuration can be laid out in the same relatively free format manner without any concerns over the case used for alphabetic characters. In addition, user comments can be included in the configuration file as they are in a problem file.

F.2 CONFIGURATION FILE CONTENTS

A configuration file has four parts: a *channel specification*, a *clock speed specification*, a *number of taps specification*, and a *ring description part*. All parts of the configuration file are required and must appear in the order given.

F.2.1 Channel Specification

The channel specification has the form:

CHANNEL *chan_num*

where *chan_num* is a 5-character channel number. This tells OASiS which of the MicroVax's serial communication channels the OAS hardware is connected to. At the present time, the sieve uses 'TXF2:'.

F.2.2 Clock Speed Specification

The clock speed specification has the form:

CLOCK *divide_by*

where *divide_by* is an integer in the range 3 to 10. The OASiS hardware divides its base clock frequency by this value to generate the high speed clock signal used during sieving; hence, decreasing this value increases the rate at which sieving takes place. Currently, the base frequency used by the OAS hardware is 40MHz.

Under normal circumstances, the user will want to run problems at the highest possible rate by setting *divide_by* to 3. However, there may be times when a slower clock speed is needed. One such instance is during the debugging of hardware faults, when a slower clock speed makes it easier to follow the operation of the hardware using an oscilloscope or logic analyzer.

F.2.3 Number of Taps Specification

The number of taps specification tells OASiS how many values are tested simultaneously by the OASiS hardware on each clock cycle. It has the form:

TAPS *num_taps*

where *num_taps* is a positive integer value.

At the present time, the only valid value for this specification is 16, meaning all available hardware taps are used to achieve maximum throughput. However, future versions of OASiS may allow other values to be used to permit multiple problems to be run on the sieve simultaneously, or to permit the OASiS software to be used with sieve hardware that supports a different number of taps. This specification is intended to allow existing configuration files to be used by such systems without modification.

F.2.4 Ring Description Part

The ring description part specifies the rings that are available to problems in the current hardware configuration. For each ring, there is a *ring description* of the form:

RING *ring_num* MAXSIZE *ring_size*

which identifies the number of the ring in hardware and the size of the largest congruence that it can contain.[1] To match the limitations of the OAS design, *ring_num* must be an integer in the range 0 to 31 and *ring_size* must be an integer in the range 2 to 32 768. The ring description part must contain at least one ring description. Since the ring number in a ring description must be distinct from all other ring numbers, there is an implicit upper bound of 32 ring descriptions allowed.

Important Note. Although the above specification allows a configuration to be defined in which there is a mixture of maximum sizes for the rings, and the OAS architecture is perfectly capable of implementing such a configuration, OASiS is not yet able to do an adequate job of assigning congruences to rings which differ in size. Consequently, the system currently insists that all rings *have the same maximum size.* If OASiS detects a configuration with rings of more than one size, the problem using the configuration is terminated with an error message. Since the current version of the OAS has 16 identical rings, this restriction is of little importance at the present time and will almost certainly be removed during any future system upgrade.

F.3 CONFIGURATION FILE GRAMMAR

The following page contains a Backus-Naur form description of the OASiS configuration file syntax. Non-terminal symbols are enclosed in '< >', while terminal symbols are character strings which are not enclosed in '< >'. Alternatives are separated by '|'. Terminals may be entered in either upper or lower case.

"Whitespace" (blanks, tabs, and newlines) may occur between the terminals and non-terminals making up a non-terminal, except in those non-terminals denoted with '†'. Whitespace is not permitted within a terminal. User-supplied comments begin with '!' and run up to the next '!', newline, or end-of-file. Comments may appear anywhere that whitespace may appear.

---

1 Actually, this indicates the number of 16-bit words of memory that are available in the ring's memory. It is possible to load a congruence whose modulus is up to 16 times the value of *ring_size* in circumstances when the greatest common divisor of the modulus and 16 is greater than one. However, this rarely occurs since most problems use congruences which have a modulus which is prime; thus, the practical limit for the largest modulus that can be loaded remains *ring_size*.

<config file>	::=	<channel> <clock speed> <number taps> <ring part>
<channel>	::=	CHANNEL <channel name>
<channel name>†	::=	<alphanumeric> <alphanumeric> <alphanumeric> <alphanumeric> :
<clock speed>	::=	CLOCK <integer$_2$>
<number taps>	::=	TAPS <integer$_2$>
<ring part>	::=	<ring description> | <ring description> <ring part>
<ring description>	::=	RING <integer$_2$> MAXSIZE <integer$_5$>
<integer$_n$>†	::=	<digit> <integer$_{n-1}$> | <digit>
<integer$_0$>	::=	<empty>
<identifier$_n$>†	::=	<alphanumeric> <identifier$_{n-1}$> | <alphanumeric>
<identifier$_0$>	::=	<empty>
<alphanumeric>	::=	<alphabetic> | <digit>
<alphabetic>	::=	A | B | ... | Z | a | b | ... | z
<digit>	::=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<empty>	::=	

Appendix G : A Sample Problem

This appendix contains an example of a sieving problem that was solved using OASiS. It illustrates the format of a problem file, a filter program, how a problem is submitted for execution, and the results that are generated.

G.1 PROBLEM FILE

The following problem file instructs the sieve to find the first 5 pseudocubes on the first 10 primes (see §7.2 of "Some OASiS Results"). This file is referred to as PCPROB10.SIV in the remainder of this appendix.

```
PROBLEM PseudoCubes10

CONGRUENCES
!  1 !              2   :    1   ;
!  2 !              9   :    1   8   ;
!  3 !   NONRING(  5   :    1:4 ; )
!  4 !              7   :    1   6   ;
!  5 !   NONRING(  11  :    1:10 ; )
!  6 !              13  :    1   5   8   12  ;
!  7 !   NONRING(  17  :    1:16 ; )
!  8 !              19  :    1   7   8   11  12  18  ;
!  9 !   NONRING(  23  :    1:22 ; )
! 10 !   NONRING(  29  :    1:28 ; )

PARAMETERS
         RECORD FROM 0
         SOLUTION LIMIT = 5
         FILTER = '[OASIS.PROBLEMS.FILTERS]NONCUBE'
         FILTER LIMIT = 1 MINUTES
```

It should be noted that most sieve problems are considerably longer than this simple example. Also, most problem files do not need to override OASiS' default ring loading algorithm using the RING or NONRING contructs.

G.2 FILTER ROUTINE

The filter routine used by the above problem consists of a pair of routines: one which determines if its multi-precise argument is not a perfect cube, and the other which calculates the integer part of the cube root of a multi-precise integer. These are linked with a precompiled mainline routine (not shown) to form a complete filter program. The mainline is a simple loop that accepts a multi-precise integer from OASiS, invokes the routine 'filter', and returns its result to OASiS. Note that the name of the user's filter routine *must* be 'filter' for the mainline to call it correctly.

```
#include mp.h
extern    MINT      *mp_curt() ;       /* forward declaration */

/*
 *        Filter tests makes sure its argument is not a perfect cube.
 *        Returns    0 if acceptable
 *                   1 if not acceptable
 *                  -1 if an error occurs during testing
 */

int       filter(n)
MINT      *n ;
{
          int       res ;     /* result of function  */
          static    MINT      cubeRoot = {0,NULL} ;
          static    MINT      cube = {0,NULL} ;

          /* return 'not acceptable' for test by Vax software (0) */
          if (n->len == 0)  {
                    res = 1 ;
                    goto done ;
                    }

          /* return 'not acceptable' if number is a perfect cube */
          mp_curt(n,&cubeRoot) ;
          mp_mult_pos(mp_mult_pos(&cubeRoot,&cubeRoot,&cube),
                    &cubeRoot,&cube) ;
          res = mp_eq(n,&cube) ? 1 : 0 ;

          /* return 'failure' if multi-precise error has occurred */
          if (mp_error())
                    res = (-1) ;

done:     return(res) ;
}
```

(Continued next page)

```
/*
 *      Sets 'roota' to the cube root of 'a' (or integer part thereof).
 *      Returns 'roota', or NULL if error.
 *
 *      Generates cube root using Newton's method.
 *      ie. x[i+1] = x[i] - (x[i]**3-a)/(3x[i]**2) where x[0] = a.
 */

MINT    *mp_curt(a,roota)
MINT    *a,*roota ;
{
        MINT    xold,xnew,xsqr,xcub,absa ;
        short   one = 1 ;          MINT    mpOne = { 1,&one } ;
        short   three = 3 ;        MINT    mpThree = { 1,&three } ;

        /* check for null arguments */
        if (mp_null(a) || mp_null(roota))    roota = NULL ;

        /* handle 0 as a special case (avoids division by 0 in loop) */
        else if (a->len == 0)      roota = mp_free_val(roota) ;

        else    {
                mp_init_val(&xold,0) ;     mp_init_val(&xnew,0) ;
                mp_init_val(&xsqr,0) ;     mp_init_val(&xcub,0) ;

                /* for simplicity, find root of abs(a) */
                absa.val = a->val ;
                absa.len = (a->len >= 0) ? a->len : -a->len ;

                /* use Newton's method to find cube root */
                mp_dupl(&absa,&xnew) ;
                do      {
                        mp_move(&xnew,&xold) ;
                        mp_mult_pos(&xold,&xold,&xsqr) ;
                        mp_mult_pos(&xsqr,&xold,&xcub) ;
                        mp_sub_pos(&xcub,&absa,&xcub) ;
                        mp_mult_pos(&xsqr,&mpThree,&xsqr) ;
                        mp_div_pos(&xcub,&xsqr,&xnew,&xcub) ;
                        mp_sub_pos(&xold,&xnew,&xnew) ;
                        }
                while (!mp_error() && mp_ne(&xnew,&xold)) ;

                /* loop may stop too high, so go down until correct */
                while (!mp_error() && mp_gt(
                        mp_mult_pos(mp_mult_pos(&xold,&xold,&xsqr),
                        &xold,&xcub),&absa))
                        mp_sub_pos(&xold,&mpOne,&xold) ;

                /* fix up answer if argument was negative */
                if (a->len < 0)   xold.len = (-xold.len) ;

                mp_free_val(&xcub) ;       mp_free_val(&xsqr) ;
                mp_free_val(&xnew) ;
                roota = mp_error() ? NULL : mp_move(&xold,roota) ;
                mp_free_val(&xold) ;
                }
        return(roota) ;
}
```

G.3 RUNNING THE PROBLEM

The following shows PCPROB10.SIV being submitted to OASiS for execution. User input is shown in **bold**. Prior to this, the user had signed on to the MicroVax as "OASIS" and set the default directory to the directory containing the problem file.

```
$ qjob pcprob10
Use default sieve configuration? (Y/N): y
Use default sieve job queue? (Y/N): y
Use default job priority? (Y/N): n
Job Priority (Default = 50): 75
Job PCPROB10 (queue SYS$OASIS, entry 360) pending
   pending status caused by queue busy
$ [next command ...]
```

The problem begins execution when it reaches the top of the SYS$OASIS job queue. A VMS mail message is sent to username OASIS when the problem terminates.

G.4 PROBLEM RESULTS

After execution, PCPROB10.SIV contains a log of the problem results, including solutions, checkpoints, and error messages.

```
PROBLEM PseudoCubes10

CONGRUENCES
        [as before]

PARAMETERS
        [as before]

LOG
    0 00:00:01    MESSAGE  Problem file read successfully
    0 00:00:01    MESSAGE  Mon Jul 17 14:44:52 1989
    0 00:00:01    MESSAGE  Initializing sieve system devices
    0 00:00:13    VERIFIED 0        SOLUTIONS TO      1
    0 00:00:15    SOLUTION 1        AT       2393
    0 00:00:15    SOLUTION 2        AT       3457
    0 00:00:15    SOLUTION 3        AT       3583
    0 00:00:16    SOLUTION 4        AT       4789
    0 00:00:16    SOLUTION 5        AT       5669
    0 00:00:16    MESSAGE  Solution limit reached
    0 00:00:16    MESSAGE  Verifying sieve contents
    0 00:00:18    VERIFIED 5        SOLUTIONS TO      5671
    0 00:00:19    MESSAGE  Problem execution terminated
```

The file PCPROB10.LOG is also produced, and shows the commands executed by the batch job which ran the problem, as well as a summary of the resources it used. The portion of the file shown below has been modified to remove lines that are not related to OASiS and to improve readability. Lines beginning with '$!' are comments.

```
$!
$! This command proc is always run when anybody on the entire system
$! logs in. It is equivalent to LOGIN.COM except that the instructions
$! contained herein are executed everytime anyone on the VMS system
$! logs in to their account.
$!
[many lines omitted]

$! Define new commands
$ kill   == "$ sys$user:[oasis.software.killer]killer"
$ priv   == "set proc/priv=(sysprv,phy_io,detach)"
$ sieve  == "$ sys$user:[oasis.software.monitor]monitor"

$! Set up privileges so that 'sieve'and 'kill'commands work properly
$ priv
$ define/table=lnm$process_directory lnm$temporary_mailbox lnm$group

$! Set job priority to level 3 to avoid competing with interactive users
$ set proc/priv=altpri
$ set proc/prio=3

$! Allow job to continue if errors occur (permits message to be mailed)
$ set noon

$! Display time (for user's benefit only)
$ show time
  17-JUL-1989 14:44:51

$! Start problem execution by invoking sieve monitor program
$ sieve SYS$USER:[OASIS.PROBLEMS.PSEUDOCUBE]PCPROB10.SIV;

$! Job done - mail message to user
$ create dummy.txt
Problem file SYS$USER:[OASIS.PROBLEMS.PSEUDOCUBE]PCPROB10.SIV; has
completed execution
$ mail/subject=jobdone dummy oasis
$ delete dummy.txt;

  OASIS        job terminated at 17-JUL-1989 14:45:17.66

  Accounting information:
  Buffered I/O count:          2686      Peak working set size:   776
  Direct I/O count:             134      Peak page file size:    3998
  Page faults:                 1357      Mounted volumes:           0
  Charged CPU time:     0 00:00:13.84    Elapsed time:     0 00:00:41.50
```