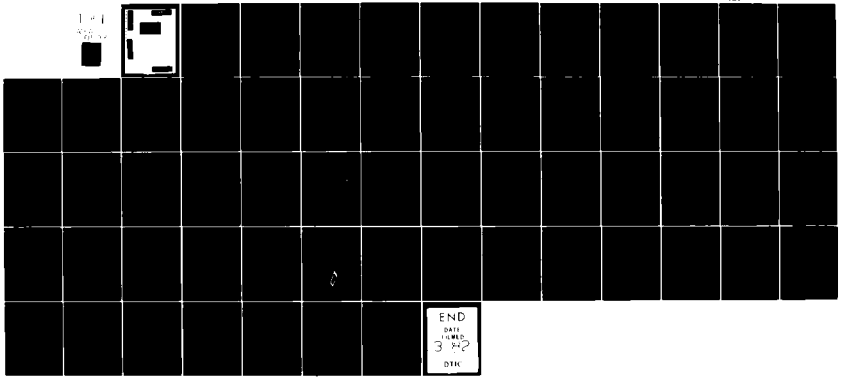
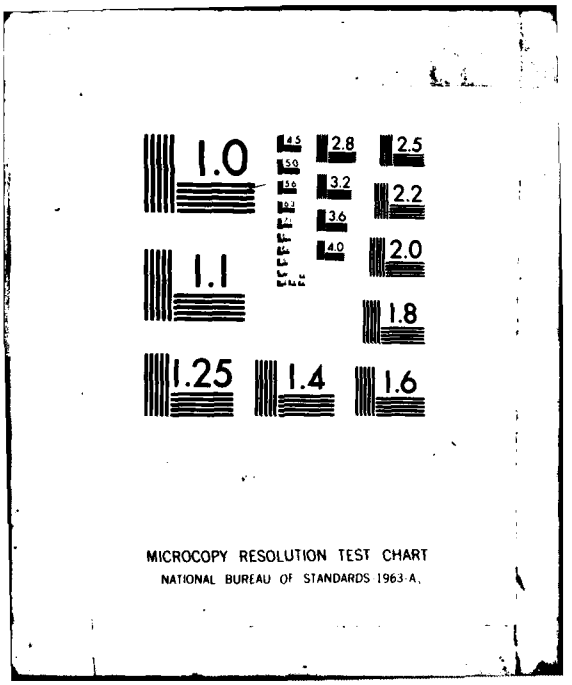
AD-A110 593 RENSSELAER POLYTECHNIC INST TROY NY DEPT OF CIVIL EN--ETC F/G 9/2
THE IMPLICATIONS OF VLSI ROM CHIPS ON NUMERICAL ANALYSIS.(U)
JAN 82 L J FEESER, M F ROONEY, M S SHEPHARD N00014-80-C-0712

UNCLASSIFIED

NL



END
DATE FILMED
3-82
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A.

(3) LEVEL II

The Implications of VLSI ROM Chips on Numerical Analysis

January 8, 1982

Principal Investigators: L. Feeser
M. Rooney
M. Shephard

Agency: Office of Naval Research

Contract No.: N00014-80-C-0712

R. P. I. Project No.: 5-24350

TABLE OF CONTENTS

The Implications of VLSI ROM Chips on Numerical Analysis

Introduction

Very large scale integrated circuits (VLSI) are an established technique for implementing computing systems and subsystems. This technology has resulted in both increased processing capability and reduced hardware cost to a scale that custom processing elements are practical for small to medium application areas. In particular, the goal of this research effort is to focus upon applications in numerical analysis and structural engineering computations.

In an earlier technical report [1], the hardware aspects of VLSI technology were explored. A series of reports each focusing upon the application and implementation of one VLSI hardware feature now follow. This first report concentrates upon read-only memory (ROM) circuits and their derivatives. The report is divided into four sections: overview of VLSI advances, a brief review of ROM circuits, storage of data on ROM chips, and storage of software on ROM chips (or firmware).

This report will show the potential speed increases made available by storing pre-computed data on ROM rather than computing upon request. The report will also demonstrate the availability of customized processors,

particularly micro-computers, made possible through firmware. The implementation of these ROM applications will be discussed with emphasis on potential changes to current programming practice, a prime concern of the numerical analyst.

Overview

While this section is not crucial to the comprehension of this report, it is important to maintain a perspective on the advances in micro-electronics with the intent of isolating trends and predicting future cost/performance ratios. This section, therefore, presents a brief overview of those advances.

There are two ways to view the advances resulting from large scale integration of transistor circuits on silicon wafers [2]. First, for a constant cost, VLSI has resulted in increased performance. Second, for a constant performance level, processing costs have decreased. In reality, these are simply restatements of the same concept, shown graphically in Fig. 1. The diagonal lines represent points of equi-potential performance. These have historically remained straight, and it is predicted that these trends will continue.

As a result of performance/cost increases, the levels of integration have also progressed, following the trend illustrated in Fig. 2. Currently, the IBM 370/168 has been

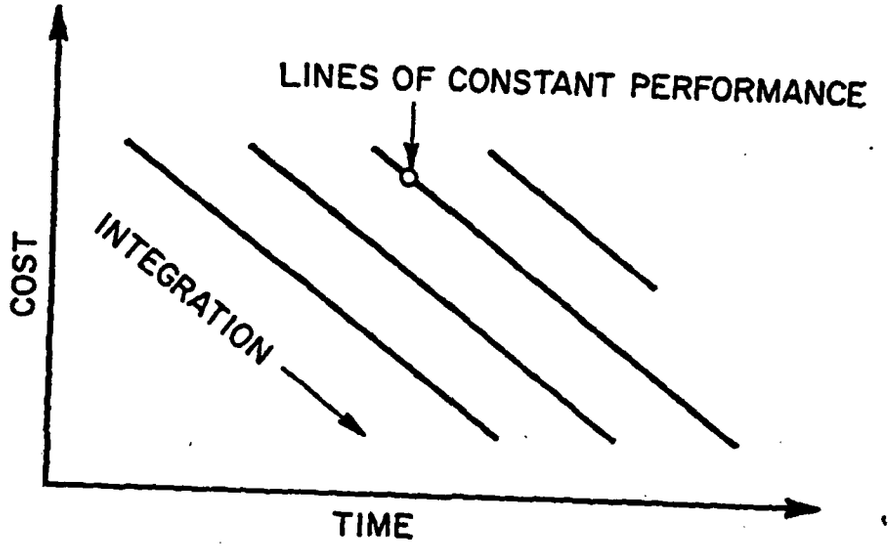


Fig. 1 - Trends in Integration

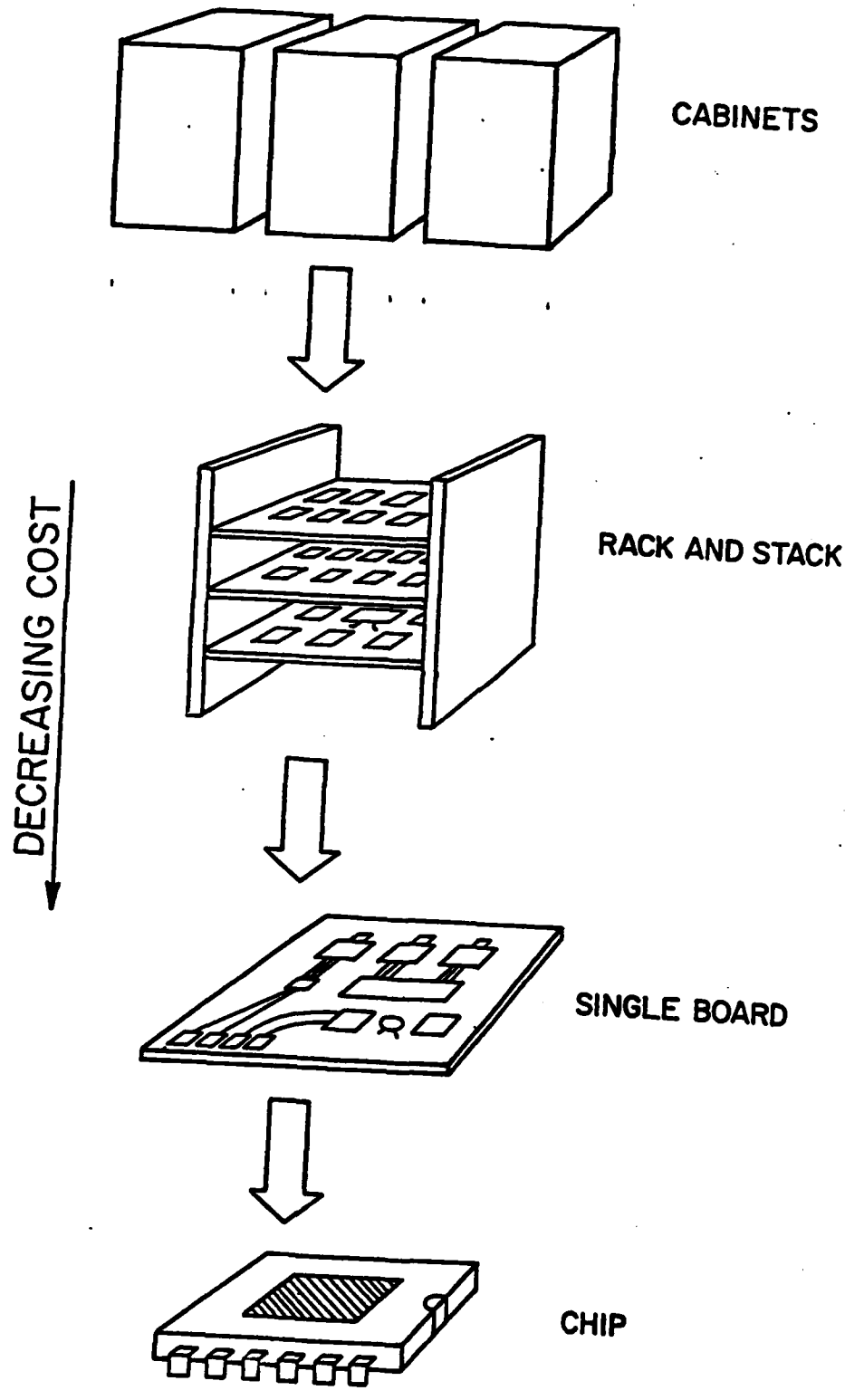


Fig. 2 - Trend of Integration

converted to an experimental chip; 16 and 32 bit micro-processors, some with virtual memory, are beginning to appear. And, research is underway to implement a DEC-VAX system onto a single wafer. Memory circuits are following the same pattern of increased integration.

VLSI technology can only be applied to transistor circuits, not electro-mechanical devices. As a result, the primary influences are felt in: CPU's, memory circuits, peripheral electronics (e.g., I/O drivers), and improved reliability. Secondary influences affect power supply circuits, packaging size reductions, and software. Little or no influence exists for electro-mechanical devices such as printers and tape/disk transports.

With these trends of improved performance/cost ratios in mind, the remainder of this report is devoted the examining the impact of readily available read-only memory (ROM) circuits.

Review of ROM Hardware

While the first technical report [1] examined the hardware aspects of VLSI technology, including read-only memory; a brief review of the basic operation of ROM chips is included for completeness and some historical perspective is added. More importantly, the definitions of "ROM", "PROM", and "EPROM" used throughout this report are presented in this section. Finally, it is shown that

current size limitations of ROM chips can be easily overcome by using several chips.

Read-only memory is defined as any type of storage system whose contents can be read, but not changed by the computer. One simple form of secondary read-only memory is the punch card, which actually preceded the first computer. However, it is the intent of this report to focus only upon primary ROM storage; that is, circuits which are part of the CPU hardware and can be randomly accessed.

Ironically, primary ROM circuitry was the original method for programming electronic computers. The "programmer" was required to construct a circuit by plugging jumper wires into a pegboard as shown in Fig. 3. The entire board was then plugged into the computer and the wires (hopefully) made the proper connections. Needless to say, the programming process was tedious, but once constructed, it made the CPU a custom processor. John Von Neumann changed the procedure by storing and handling programs like all other data, and ROM circuitry all but vanished except for bootstrap mechanisms.

The need for ROM circuitry remained, but its usage was hindered by high cost and large physical size as well as the complexity of constructing it. By applying the VLSI technologies to memory circuits, as done for processor elements, ROM circuits are realizable for the storage of

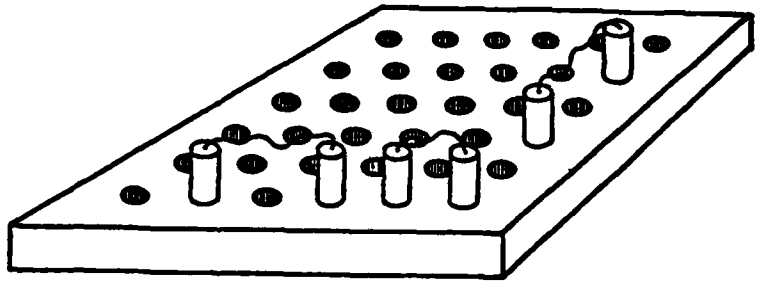


Fig. 3 - Pegboard ROM

programs and frequently used data. In fact, the highly repetitive nature of memory cells has resulted in even greater cost reductions for ROM than for processors.

A typical configuration for a read-only memory chip is shown in Fig. 4. A binary coded address enters on the left. The address is decoded and closes the appropriate internal switches to allow the contents of the specified memory location to flow out on the right through the data lines. The operation of each memory cell is still based upon the pegboard/jumper system. However, there are three major commercially available systems with different techniques for creating the jumpers.

In a conventional ROM (usually denoted by only the letters ROM), the jumpers are created and not created by the pattern transferred onto the silicon wafer. That is, they are manufactured directly. The advatage of this approach is extremely low cost when produced in quantity. The drawback is that a large number must be produced to offset the high fixed manufacturing costs (e.g., making the masks or patterns).

The programmable read-only memory (denoted as a PROM) is the next level of sophistication. The chip is manufactured without any data in it. After physical manufacture, the virgin chip is placed in a programming device and the data is programmed into it. The final PROM

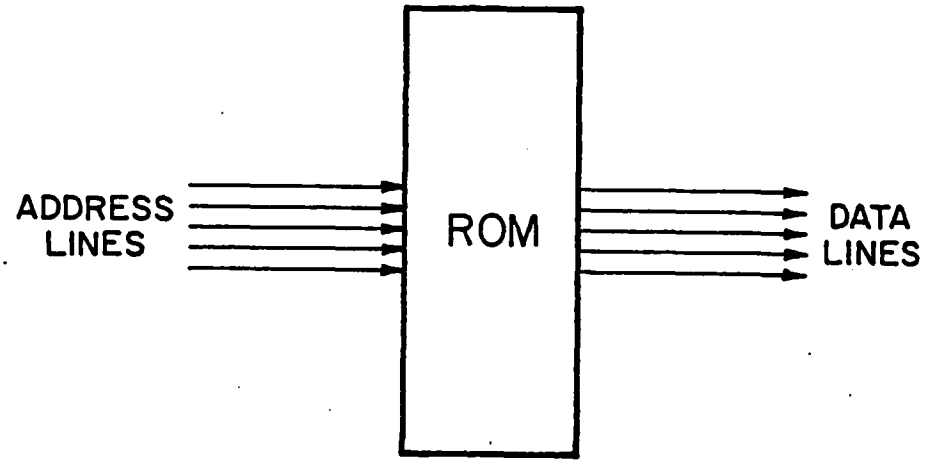


Fig. 4 - Typical ROM Chip Configuration

is then used like a conventional ROM. This programming can be done only once, however. Fig. 5 illustrates a simple model of the internal workings of a PROM and is presented to aid in understanding the limitation (i.e., irreversibility) of PROM circuits. Each data line in each memory location is connect to the supply voltage on one end, to the external data lines and addressing switches on the other end and contains a segment of reduced diameter "wire". This reduced segment acts as a fuse element. Under normal working conditions, the supply voltage is low enough that the induced current through the fuse section of the data line does not cause it to burn out. In order to program a new chip (one where all fuses are intact), the supply voltage is raised. When any data line is connected to ground, sufficient current flows through the data lines to burn out the fuse. If a zero (no voltage) bit is desired, the fuse is burned out; and vice versa. It should now be clear that once a zero bit is "burned in" to a PROM, it cannot be changed. The advantage of a PROM is the ability to program it after manufacture making it well suited to small quantity applications. However, the added step of programming makes the chip most costly.

At the pinnacle of read-only memory technology is the erasable programmable read-only memory (denoted as an EPROM). Both the ROM and PROM are final in their

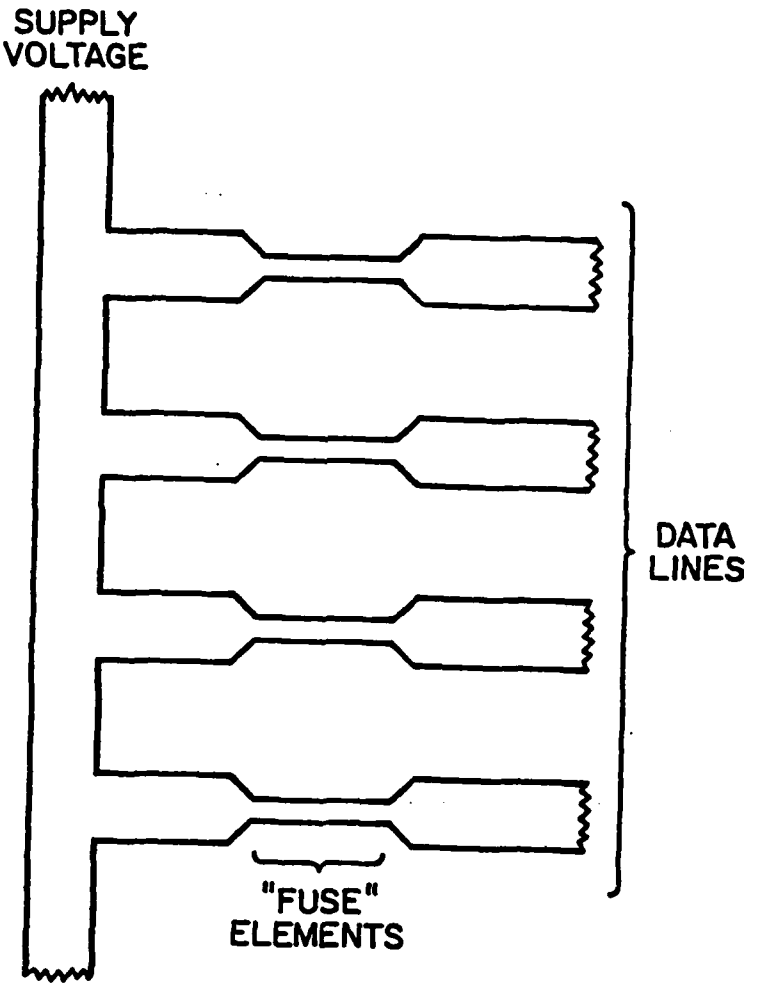


Fig. 5 - Simple Model of a PROM

programming; if an error is embedded into the chip, there is no way to correct it and the chip must be discarded. EPROM's, are similar to PROM's but use a different fusing mechanism, one that can be repaired. Repairs are usually done on a global basis, essentially creating a new chip that must be completely reprogrammed. A common arrangement uses ultraviolet light to refuse the junctions. This added feature often makes the chip more expensive; but more importantly it makes the chip vulnerable to accidental erasing. As a result, the EPROM is best suited to small volume applications where errors are probable to occur, such as an experimental environment.

While the capacity of read-only memory chips have and will continue to increase, there are many applications where the capacity of a single chip is simply not sufficient. As shown in Fig. 6, it is a simple matter to use two or more chips together. Some of the address lines are used as address inputs and are sent to all ROM chips simultaneously. The remaining address lines are fed into a switching circuit which connects the output of the appropriate ROM to the final data output lines. In this manner, the switching circuit acts as a chip select mechanism and the effect is to create a much larger ROM chip.

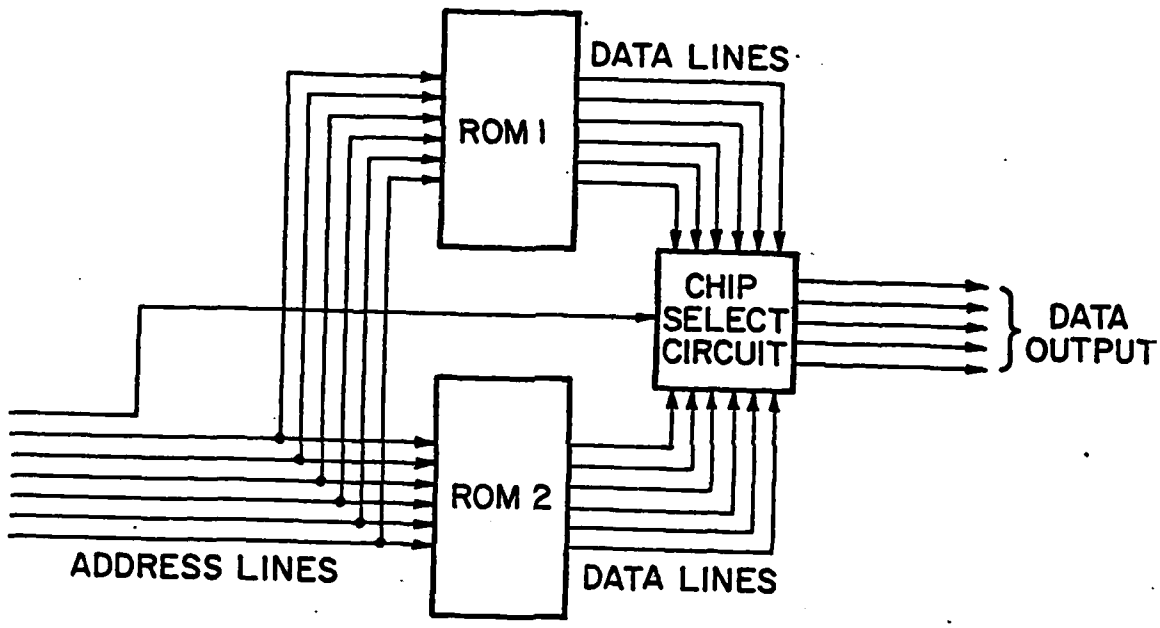


Fig. 6 - Extending ROM Size

Data Storage on ROM

The storage of data is the most logical use of ROM chips. This section examines a technique called "hardware look-up" and how it can be used in numerical analysis applications.

It will be shown that the technique can handle functions of integer values, functions of real values, functions of multiple values, and even simultaneously evaluate multiple functions. Two possible programming implementations will be examined; both requiring little or no changes to existing programs and current programming practice. For the numerical analyst, the prime result will be significant speed increases, and a simulation procedure is demonstrated for predicting expected increases. The current technology will handle "hardware look-up", and the mechanics of placing data on ROM is discussed, including a multi-computer approach called "down-line loading" which enables a micro-computer to emulate a main-frame computer's capabilities.

Hardware Look-Up. The mechanics of the "hardware look-up" scheme works as follows: The input data value is written into a specific memory location. This data is then used as an address by the ROM chip. The output data value is then transferred by the ROM chip into some special memory location. The flow of data is shown in Fig. 7. The scheme

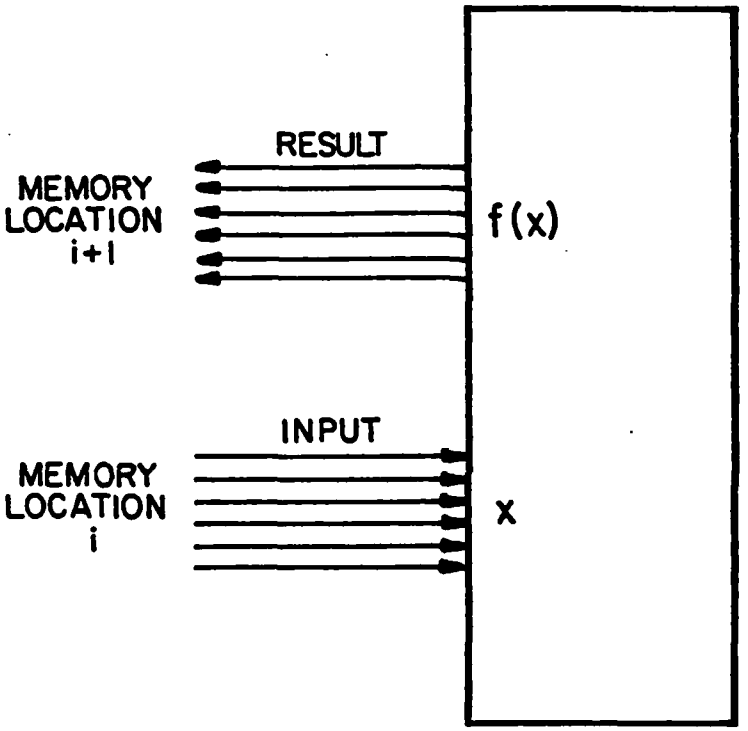


Fig. 7 - Harware Look-Up

can be extended to functions of two variables by using the combination of two input data as one address for the ROM as shown in Fig. 8. Further, several ROM chips can be connected to the same memory locations to allow several functions to be "looked-up" simultaneously as shown in Fig. 9.

ROM chips are designed for the storage of discrete data; that is, data which is accessed by whole integer numbers. Thus, we may think of data stored in ROM as elements of an array, and the input address to the ROM may be thought of as the index of the array. This makes ROM chips well suited to the storage of tables such as wide-flange beam sizes and properties. In fact, for multi-column tables (tables consisting of multiple items per entry line), an arrangement of parallel ROM chips is ideal, for it allows quick access to all entries and simplified installation of updates.

Most of the data and/or functions used by the numerical analyst are continuous or analog in their nature. That is, they have an infinite number of values. Because ROM chips are discrete (i.e., have a finite number of storage locations), it is <u>theoretically</u> impossible to store a continuous function. However, by considering a practical limitation of computers, numerical round-off, it is not only possible but actually quite simple to store a continuous

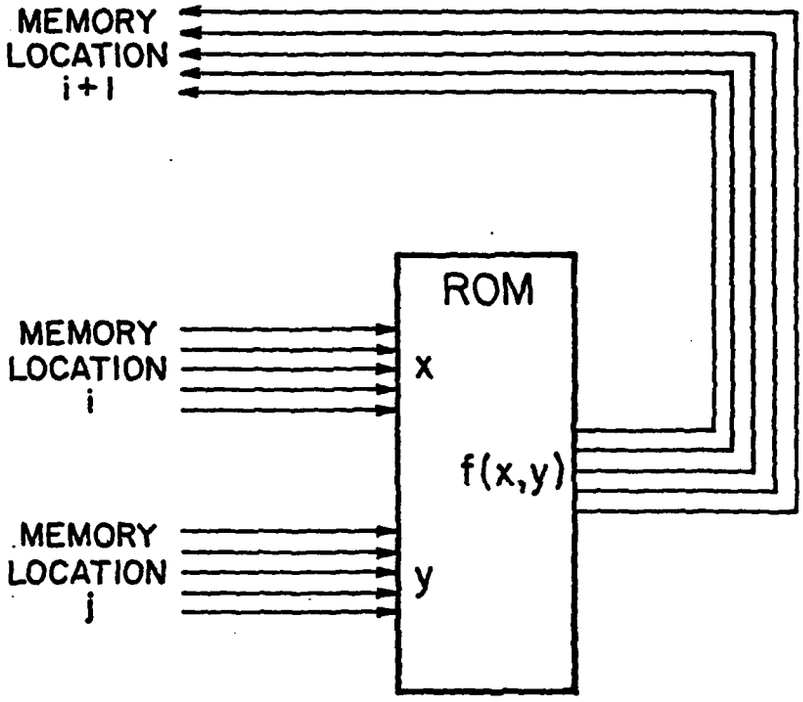


Fig. 8 - Extended Hardware Look-Up

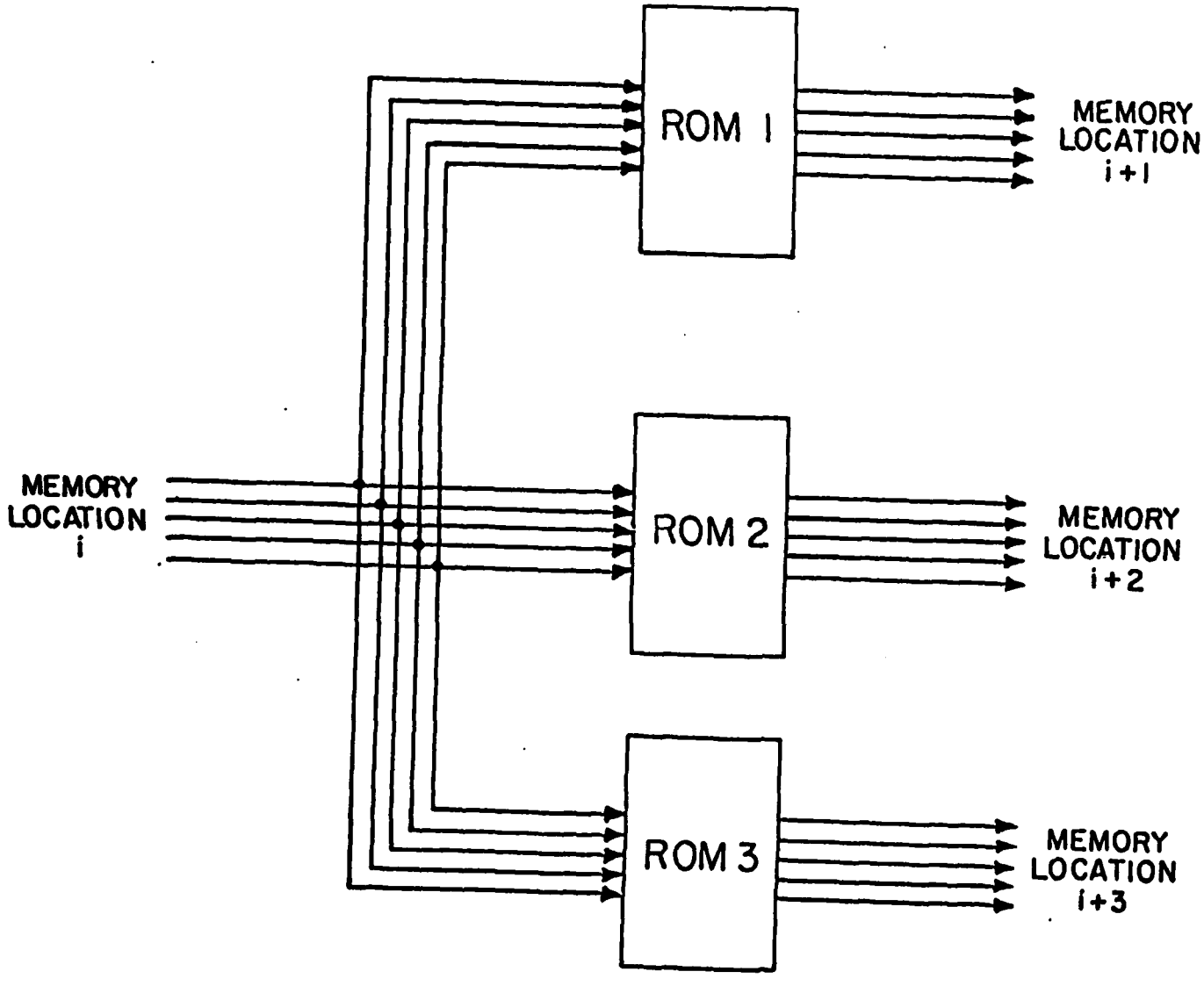


Fig. 9 - Parallel ROM Look-Up

function.

All numbers, integer or real, are stored in the computer as a pattern of binary digits [6]. The number of digits used is determined by the word size of the computer and the encoding scheme, but is a fixed number for a given type. However, for a fixed number of binary digits, say n, only 2**n possible permutations exist. In order to accommodate real numbers using a fixed number of digits (and thus a finite number of permutations), the desired range of real numbers is divided into subregions and each subregion corresponds to a permutation. (See Fig. 10.) Any real number within a given subrange is assigned to the same permutation, and thus are all treated as the same real number. This process is conventionally called "round-off". The important result is that the continuous set of real numbers have been converted to a discrete set of binary permutations. By interpretting the binary permutation patterns as integers, an address can be obtained for storage and retrieval on the ROM chip. (The net result is much like that of printing a real number with an I format in the FORTRAN language.)

In summary, real numbers are encoded to a discrete set of bit patterns through round-off; the patterns are then used as input addresses to the ROM chips. No accuracy is lost, no numbers unaccounted for, as all storable real

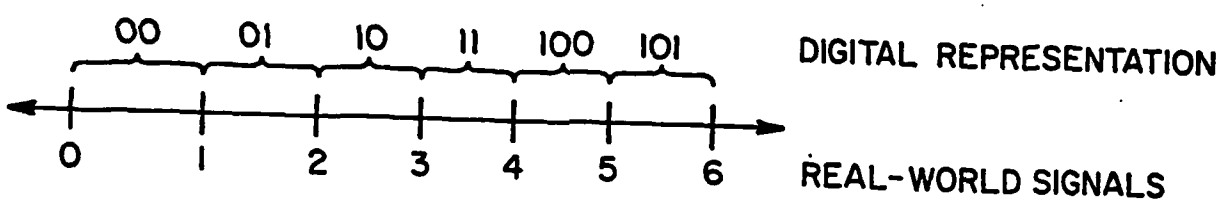


Fig. 10 - Discrete Nature of Digital Signals

numbers are used. Thus, continuous furnctions (such as sine, logarithms, and polynomials) may be computed once and completely stored on ROM by storing the corresponding values for all of the storable real input values. For a 16 bit encoding scheme, 64K of ROM is required.

Programming Interface. To the numerical analyst, the most important issue is how ROM chip capabilities will be used with programming languages; and more specifically, how current programs will have to be modified to take advantage of the new technology. The discussion presented here is geared toward the FORTRAN language; however, similar constructs are available in most other languages, most notably, BASIC, the language used on most micro-computers. Two major approaches are available: function calls, and special COMMON blocks.

The COMMON block approach requires the programmer to include a special labelled COMMON block in the program. The linker (or linking loader) is used to bind this block to the locations in memory where the ROM circuits reside. Most linkers presently provide this capability in their advanced or extended features. The ROM chips may then be used in the manner previously described: input data is sent by making an arithmetic assignment, and the generated output data is available by simply using the appropriate variable from the common block. For example:

```
COMMON /TRIG/ X,SIN,TAN
X=Y*3.14/180.
Z=SIN+(2.*TAN)
```

The first line designates the special COMMON block; the second line loads a value into the parallel ROM circuits; and the third line uses two of the output data values.

The function call approach behaves the same as any system or language supplied function. That is, an ordinary function call is made (e.g., X=SIN(Y)), and a machine or assembly language routine is involked to generate and return the proper value. Further, an array is syntactically called in the same manner as a function; thus, the function call approach can also be compared with addressing arrays. The blending of FORTRAN and assembler routines is done with the linker (or linking loader) and can be easily handled by all existing linkers.

Both approaches are quite workable and simple. The COMMON block approach is better suited when several inputs are needed, and is required if several output are generated by a parallel ROM arrangement. This latter condition occurs because a FORTRAN function can return only one value. The function call approach is vastly superior for existing software. Present code can be left intact: the code for the function is simply replaced with new function code which accesses the ROM chips. However, the COMMON block approach

can be converted easily to a function call approach (i.e., the COMMON blocks can be hidden away in the function call). As a result, the COMMON block technique should be used as the implementation scheme and the programmer can then use both.

The most significant result is that current programming practice can continue, and conversion can be done to existing code without rewriting it.

Speed Increases. The primary benefit from hardware look-up will be increased speed. This increase will be derived from pre-computing all potentially needed values rather than generating them upon request, much like mathematical tables are (or were) used in engineering practice. In order to gain some insight into the magnitude of the increase, a simple simulation procedure was carried out.

The procedure consisted of running a pair of programs for several different functions. The first of the pair of programs contained a conventional function call; thus, computations were performed upon request. The second program of each pair simulated the ROM chips by: 1) creating an array, 2) computing all potential values of the function and storing them in the array, 3) changing all function calls to array accesses (although this actually requires nothing as the syntax is the same for both), and 4) running

the program. The comparison was made by monitoring the CPU time (through system subroutines linked to the operating system) of the complete first program and the execution after loading the array for the second program. Each pair of programs were run four times to minimize system loading fluctuation and the results are shown in Fig. 11. A typical program pair is shown in Fig. 12, and the computations were performed on an IBM 370/3033.

Figure 13 graphically depicts a series run on polynomials. The gradient change at sixth-order polynomials is believed due to a change in computational procedure: specifically, changing from repetitive multiplication to a logarithm-antilogarithm scheme.

Two important trends can be seen by examining the table of Fig. 11 and the graph of Fig. 13. First, for standard mathematical functions (e.g., SIN, LOG), speed increases of 15 to 20 can be realized. These savings can be applied to a large range of numerical computations (e.g., development of rotation matrices and fourier transforms). Second, for custom functions such as polynomials, speed increases are related to the complexity of the function and result in speed increases of several orders of magnitude. It should be noted that no substantial speed increase will result from converting tabular data stored in arrays to ROM chips, but other benefits are derived.

	Time for function evaluation (cpu sec)	Time for array evaluation (cpu sec)	Speed Increase*
Multiplication	0.777	0.095	8.18
Addition	0.727	0.095	7.65
Division	0.835	0.105	7.95
Logarithm	1.649	0.099	16.66
Sine	1.694	0.099	17.11
Cosine	1.675	0.094	17.82
Exponentiation	1.658	0.094	17.63
Polynomials			
0^{th} order	0.585	0.101	5.79
1^{st} order	0.707	0.101	7.00
2^{nd} order	0.766	0.101	7.58
3^{rd} order	0.893	0.101	8.84
4^{th} order	0.988	0.101	9.78
5^{th} order	1.105	0.101	10.94
6^{th} order	1.243	0.101	12.30
7^{th} order	1.944	0.101	19.25
8^{th} order	2.649	0.100	26.49
9^{th} order	3.378	0.101	33.45
10^{th} order	4.108	0.101	40.67

*Speed Increase = Function Time/Array Time

Fig. 11 - Potential Speed Increases with ROM Circuits

Function Call

```
      F(K)=K**2
      CALL TIME
      DO 2Ø I=1,1ØØØ
      DO 1Ø J=1,5Ø
      Z=F(J)
1Ø    CONTINUE
2Ø    CONTINUE
      CALL TIME
      STOP
      END
```

Array

```
      DIMENSION F(5Ø)
      DO 5 I=1,5Ø
      F(I)=I**2
5     CONTINUE
      CALL TIME
      DO 2Ø I=1,1ØØØ
      DO 1Ø J=1,5Ø
      Z=F(J)
1Ø    CONTINUE
2Ø    CONTINUE
      CALL TIME
      STOP
      END
```

Fig. 12 - Typical Program Pair

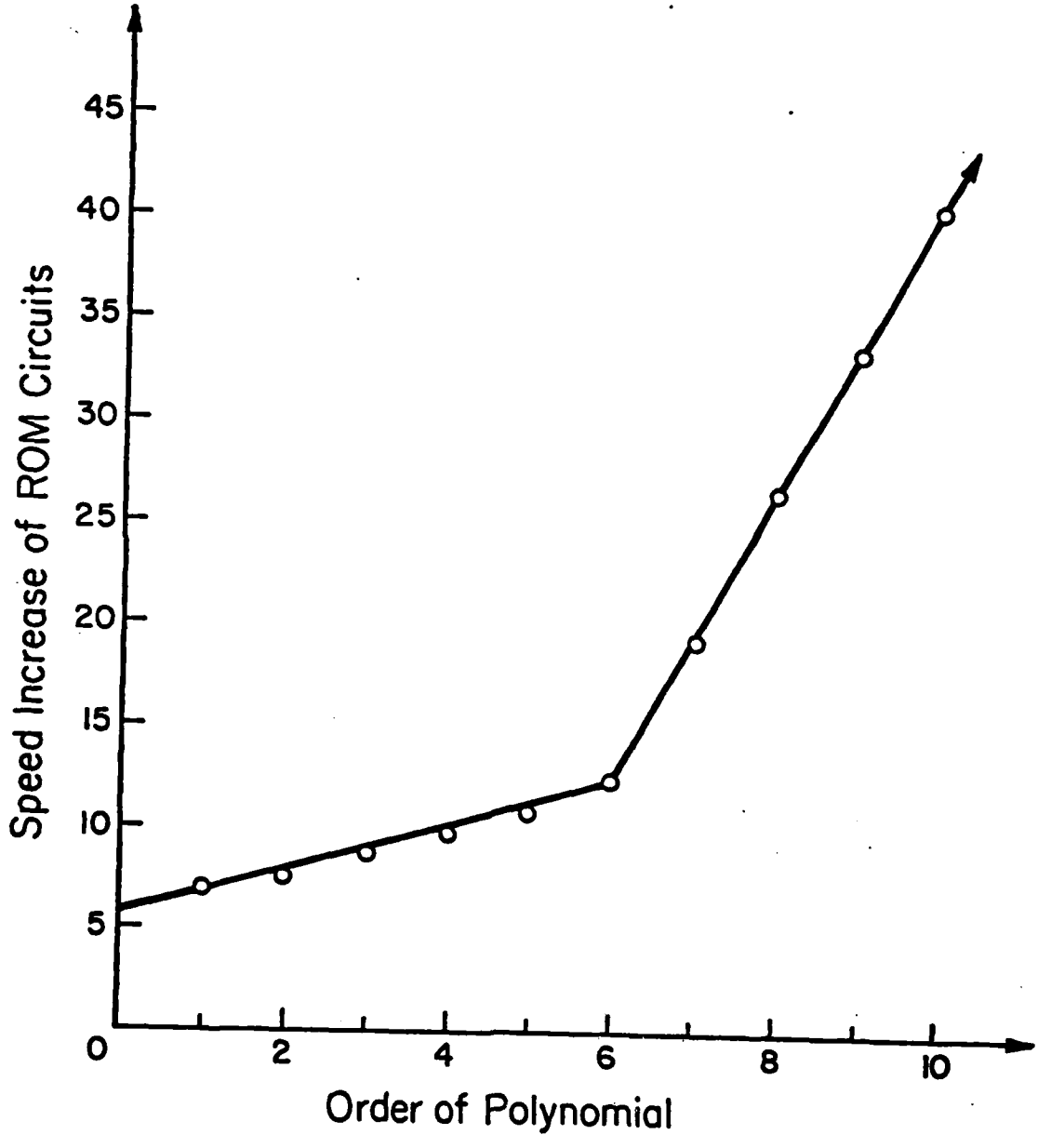


Fig. 13 - Speed Increases for Polynomial ROM Chips

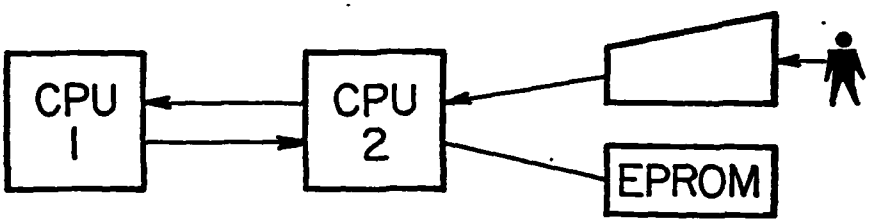
Down-Line Loading. The programming of PROM and EPROM chips is, of course, done by a computer. This manufacturing process leads to an interesting extension, particularly if EPROM chips are used. Fig. 14 outlines the major steps.

Down-line loading is a procedure where one computer programs another, regardless of the storage medium. It is suggested here that the programming be a transfer of data and that the storage medium be a PROM circuit (or preferably, an EPROM chip).

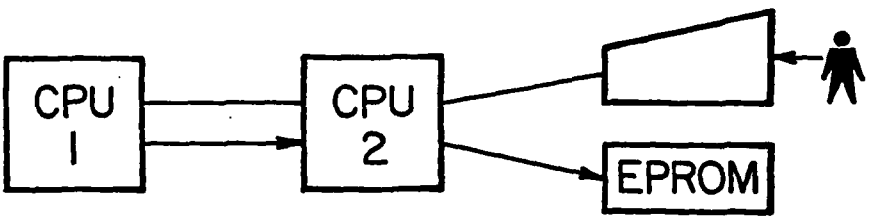
The first step is to install a virgin PROM or to erase an existing EPROM. Often the PROM must be connected to a special device during programming, one that can supply the necessary electrical signals; the completed PROM is later moved to a permanent, but ordinary, chip socket. Second, communication is established between: the two CPU's, the user and one of the CPU's, and the virgin PROM and the non-programming CPU. Third, the programming CPU transfers the data through the non-programming CPU to the PROM where the data is embedded into silicon. Communications between the CPU's is no longer needed and is usually severed. Fourth, the non-programming CPU runs alone and uses the data from the PROM as if it were initially manufactured into the CPU.

The prime advantge of down-line loading is that the power of a large computer can be used to quickly generate

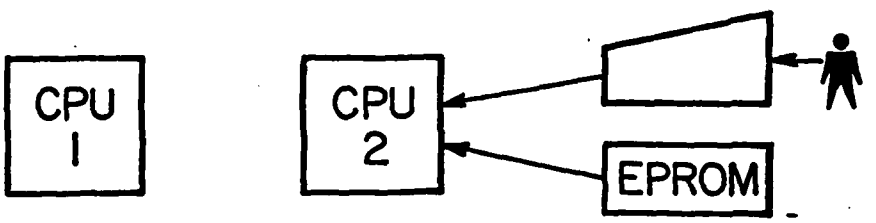
Step 1 - PROM is Erased



Step 2 - Communication Established



Step 3 - Data Sent from CPUI to PROM



Step 4 - PROM Drives Stand-Alone CPU2

Fig. 14 - Down-Line Loading

complex data to be stored on the PROM. Once programmed, the PROM can be used by a smaller machine, and thus, imitates the power of a much larger CPU. Further, because the programming is done once, the larger CPU can "train" many smaller machines. EPROM chips allows the process to be repeated as updates are needed. The communication between two CPU's is usually very fast and the smaller machine often buffers the data for the slower process of programming the PROM chip. This arrangement is best suited for a main-frame to micro-computer connection.

Needed Technology. All of the plans for placing data into ROM is presently available. But, present technology would require multiple chips in an extended ROM configuration. This would result in a somewhat expensive implementation as compared to the "dollars and cents" cost of computing upon request. The speed increases could easily tip the scales in a critical situation, however.

VLSI and ULSI (ultra-large scale integration) will soon provide a single chip capacity large enough for 16 and 32 bit single value input ROM circuits. At that point, the ROM circuit cost will be less than computing upon request.

Software Storage on ROM

Because a program is treated as data in the modern computer, ROM chips can be used to store programs and data. Further, no changes are required in the construction of the

ROM chip, and thus, the mechanics of transferring the program to the chip is the same, though the generation of the information is different. As a result, normal ROM's, PROM's, and EPROM's may all be used to store programs. The final product, software stored on ROM, is called firmware.

This section examines the impact of firmware on computing. The section is divided into five subsections, with the first three comprising the heart of the discussion. As a result of this discussion, it will become evident that the key importance of firmware is its ability to transform a general processor into a custom or semi-custom processor. For the numerical analyst, two practical applications presently exist: The first use is to extend the hardware features of the machine through firmware subroutines; these extensions, like subroutine libraries, will reduce programming cost by eliminating the rewriting of standard computational procedures. The customization will be as simple as plugging in appropriate ROM modules. The second use takes the first use an additional step and consists of putting an entire program onto ROM. This will create a complete custom processor. Current examples of these applications are shown in the last subsection, and the fourth subsection describes the creation of firmware including "down-line loading". The other applications (microcode programming and operating systems on ROM) should

not be ignored, however. They are destined to be practically available to the numerical analyst as improved sofware tools are developed.

Advantages and Myths of Firmware. The primary advantage of firmware is its ability to create a custom processor. That is, once the ROM chips are installed, the programmer (or user) may view the new software as an intrinsic part of the hardware. As we shall see shortly, these customizations may range from extended hardware features to specialize operating systems dedicated to the performance of a single task.

One peculiar advantage of firmware is that it is difficult to reproduce a ROM chip unless you are original manufacturer. Further, it is almost impossible to produce the same software for another machine if the software resides in a ROM. That is, it is a prohibitively costly procedure except for the developer of the firmware who alone has access to the source code and original manufacturing chip masks. Thus, firmware aids in the battle against software piracy. In the past, software piracy has not posed a major difficulty for the numerical analyst; but, with the proliferation of micro-computers and corresponding users, the protection of innovative programming (for the recovery of developmental costs) will become increasingly important.

Having seen the two primary advantages of firmware, we shall now expose some of the myths. It is important to firmly establish that software in ROM will NOT yield any significant speed increases. Most programs are loaded into memory prior to the start of execution by the loader or linking loader. While some speed increase certainly occurs in accessing ROM circuitry over disk, the total load time is generally negligible when compared to execution time. Some micro-computer systems appear to exhibit improved performance when operating from ROM; this, however, is an illusion. The speed increase results from executing compiled code (previously translated to machine language) instead of interpretted code (translate and executed simultaneously). This speed improvement can be obtained by simply installing a compiler program.

Firmware usage will result in some space (or memory) savings. However, as memory sizes are and will be expanding with reduced costs, the space savings will not be significant for large machines. For micro-computers the space savings are significant, but, again, an illusion. In order to facilitate firmware, a certain number of memory locations must be connected to the ROM chips; memory locations which must be robbed from normal memory. Thus, it is important to actually limit the amount of memory devoted to firmware.

Firmware has the potential to reduce the cost of software, but only when large quantities of a single item are to be produced. The potential savings result from the ability to produce an entire chip in a few operations regardless of its complexity. There are several snags in this approach, however. First, this maufacturing procedure only applies to normal ROM chips, not PROM or EPROM chips where each datum must be transferred to each chip. Second, the development cost is quite expensive, and thus, must be distributed over a large production run to make duplication a significant cost. Third, the machine targetted for by the software must be physically able to accept ROM chips and the signal interface standards must match. Because hardware varies so greatly, ROM chips are usually compatible with only one machine; thus, many of one type of machine must exist. Hence, production cost savings do not apply for main-frame computers. In fact, these cost savings are only appreciable for popular micro-computers.

Uses of ROM Software. Technically, any program or set of instructions can be placed on a ROM chip. There are, however, certain loose categories of programs that emerge. The discussion here is not meant to be complete, but rather, to indicate the scope of firmware.

The first usage is to extend or define the instruction set of a processor chip. Obviously, this requires a special

processor chip called a micro-programmable micro-processor. This processor is designed and fabricated without a fixed set of instructions. After fabrication a PROM circuit, usually part of the processor chip, is encoded to provide the processor's instruction set. The procedure is called micro-programming and uses a primative but complicated machine language called micro-code. Until such time as the micro-programing task is simplified, this process will remain impractical for numerical analysts.

The second level of usage is to permanently install program segments or subroutines into primary memory. Fig. 15 shows a typical memory map for such an arrangement. The low memory addresses store interrupt vectors and other hardware dependent data. The next low memory block holds the operating system and related data. At the high end of memory are the input/output ports to which the peripherals and peripheral control equipment are wired. The next high block of memory is reserved for ROM chips (data and software), and the ROM chips are physically wired to these locations. The remaining space is conventional primary random access memory, and can, thus, be used for pro ram and data storage. The exact distribution and order of these blocks will vary with each machine and operating system. The subroutines are encoded into ROM chips, which are then "permanently" installed into memory. (We shall explain how

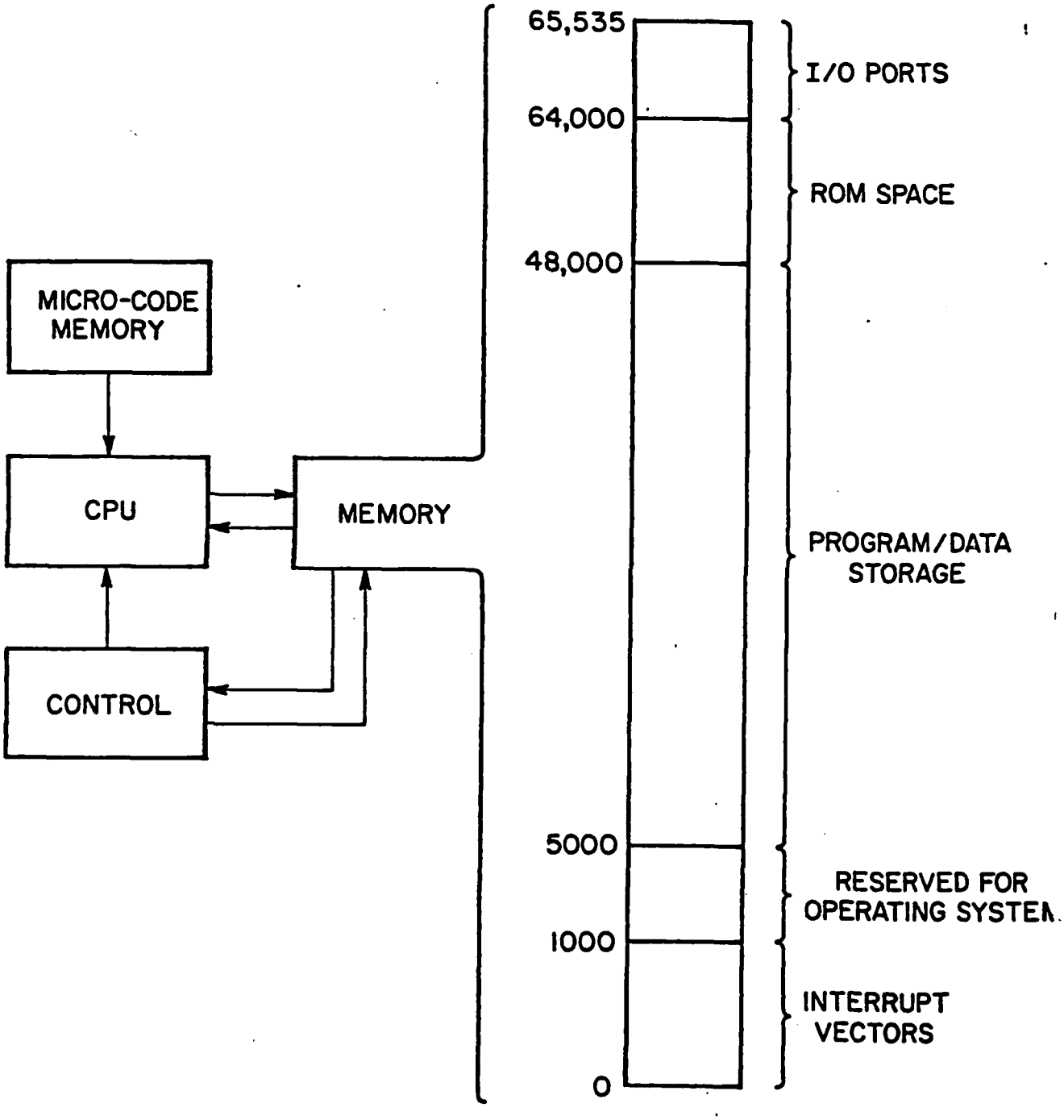


Fig. 15 - Typical Memory Configuration

they are used shortly.)

At the third level of usage, the ROM space shown in Fig. 15 is expanded and complete programs are installed. Some conventional random access memory must be retained to store the data being processed, the output data, and intermediate results. The classic Von Neumann trade-off of program storage versus data storage limits the system. As with subroutines, several programs can be resident in this configuration and selection made though the operating system.

At the fourth level of usage, the operating system can be implemented as ROM. Again, any associated data must be kept in conventional random access memory. In this fashion, the computer system becomes a dedicated processor (e.g., a word-processing machine or a finite element pocket calculator). This level of usage requires the writing of an operating system, and thus, is impractical for the numerical analyst. As software design aids become available, this technique may prove practical.

Programming Interface. A quick examination of Fig. 15 will reveal that firmware is not any different than normal software except where in memory it is stored. As a result, firmware can readily be handled with existing techniques and tools, most notably the linker, loader, or linking loader.

For the case of the complete program, no linking is required, the code is always resident, and the only operation necessary is for the loader to pass control to the appropriate starting address. The last statement in these ROM programs is to pass control back to the operating system. This procedure also applies to ROM based operating systems except the last step (returning control) is omitted as all operating systems are designed as infinite loops.

While micro-code programming is beyond the scope of this report, the use of micro-code requires the development of new compilers each time the instruction set is redefined or extended. Further, to make efficient use of the newly defined instruction set, the compilers must be optimizing compilers. Hence, it is evident that the micro-programmable processor is currently beyond the practical limits of the numerical analyst.

The only real programming interface for the numerical analyst is the use of program segments or subroutines. These subroutines are (or should be) in a machine readable form and are permanently loaded into the computer. The only concern, then, is the linking process which may be divided into two subproblems. The first subproblem is branching to and returning from the subroutines. The solution is the same as for complete programs except connections are made to a calling program rather than the operating system. Any

existing linker can accomplish this task. The second subproblem is the passing of arguments. The arguments (and other data) must reside in the random access memory, not in ROM. Thus, firmware subroutines must be compiled so that all variables reside separately from the code and all addresses be fixed. The most logical method of assuring this requirement is to use a fixed location COMMON block as was described earlier. It is, of course, possible and practical for all the firmware subroutines to share the same fixed COMMON block as firmware to firmware calls are predictable and shared COMMON blocks will save space. Other methods include passing arguments in the registers, on the system and/or user stacks, and through indirect addressing. These alternate methods are cumbersome for the numerical analyst and provide no real benefit over the COMMON block method.

Creation of Firmware. Virtually all firmware exists as machine language code, yet it is extremely rare for anyone to write machine code. Obviously, some type of translation is made from another language. At the elementary level, an assembly language is used and is translated by an assembler program. Because assembly language is very similar to machine code, assembly language affords the programmer the opportunity to create the highest speed code while simultaneously minimizing software storage requirements.

Although assembly language programs are easier to write than machine code, assembly language is still extremely cumbersome for composing large or complex programs. To alleviate the problem, a high level language such as FORTRAN or BASIC is used to write source code. This source code is then translated to machine code (or an object module) by a special program called a compiler. However, in order to support the flexibility of the high level language, a certain amount of inefficiency must be built into the compiler program. High level languages can also be translated by another type of special program called an interpretter. But, an interpretter translates one line of code at a time as the high level language is being executed. This means that every statement in a loop is translated everytime the loop is executed and the entire high level language code must be retranslated (including all the repetitions of the loops) each time the program is run. While interpretted code has the advantage that source code is stored, the extra translation time quickly outweighs the apparent advantage. As a result, interpretted code is rarely stored as firmware. One final note about translation is warranted: it is common practice to compile or assemble a program on the machine (or same model) where the machine code will operate; however, it is quite possible to compile or assemble on another machine with a special translation

program called a cross-compiler or cross-assembler. This cross translation procedure is becoming increasingly popular for the preparation of firmware.

Once the machine code has been created, it must be transferred into a chip. The mechanics for this process is exactly the same as for data; and conventional ROM, PROM, and EPROM implementations are possible. However, due to the low volumes of firmware production runs, PROM and EPROM chips are almost always used, with EPROM chips handling most of the experimental versions and PROM chips being used for final production runs. As with data storage, down-line loading is possible and used to a small extent.

Firmware Examples. Most firmware in existence (barring system bootstrap loaders) is used in micro-computer systems. As micro-computer usage grows, firmware will also expand. In this section, two systems utilizing firmware are briefly examined. These are certainly not only systems available, but are typical of what is presently done with firmware. (No endorsement of these products is implied.) The first example illustrates how firmware subroutines can be used to extend the hardware features of a machine. The second example illustrates how a complete custom processor can be created. This second example could be used to create a finite element personal computer, i.e., a machine that powers up as a finite element machine. Only the practical

limitations of memory size and micro-processor speed hinder this grand creation.

The first system is the Tektronix [3] 4050 series of micro-computers (models 4051, 4052, and 4054). A typical configuration (see Fig. 16) possesses the micro-processor, memory, a vector storage tube display, keyboard, magnetic tape drive, port for a hardcopy unit, an RS-232-C or GPIB port, and slots for ROM cartridges. The machines are programmable only in BASIC and the ROM slots are configured to provide various extensions to the BASIC language depending upon the modules inserted. Modules may be inserted, removed, and swapped by the user as his/her needs change as the modules are simply slid in and out. Presently, only Tektronix offers a line of compatible ROM cartridges and these cartridges are pre-defined (i.e., no mechanism exists for the user to generate his/her own cartridge). Typical ROM cartridges provide matrix operations and an extended line-editor. Because the firmware is in machine language, considerable speed improvements are possible over the normal BASIC language.

The second system is the Radio Shack TRS-80 Color Computer. Models range from 4K memory, standard BASIC language for $400; to 32K memory, extended BASIC for $750, both without the television monitor. In addition to BASIC, the computer can be programmed in machine language and

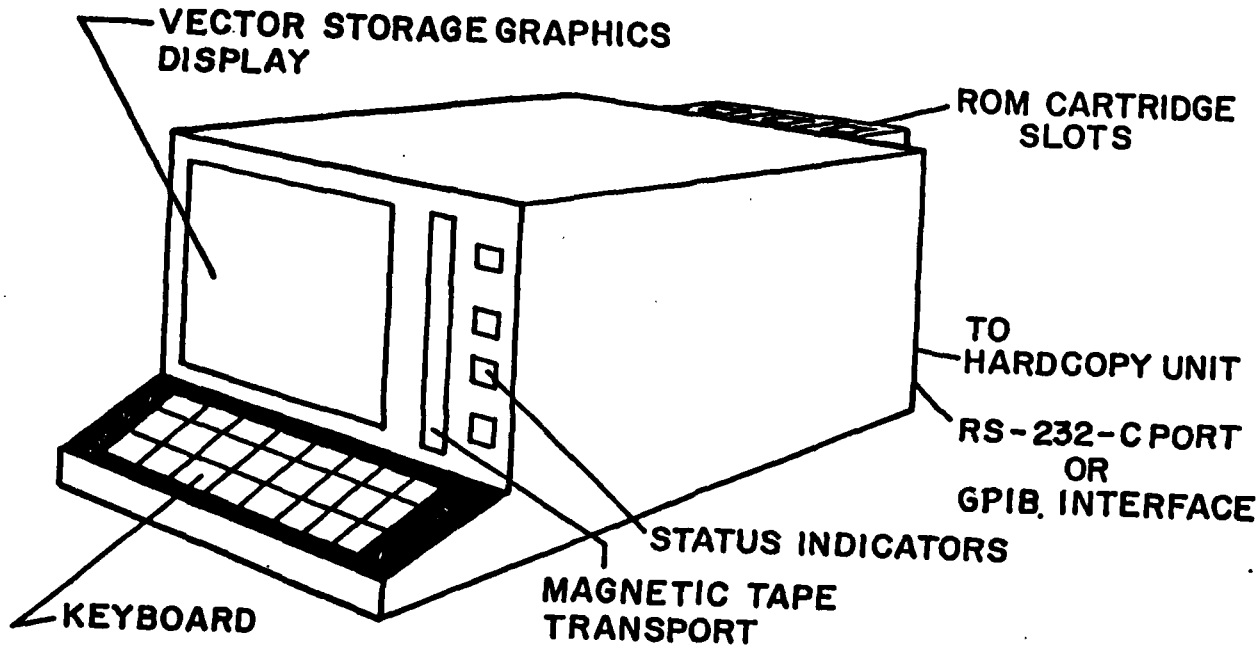


Fig. 16 - Layout of Tektronix 4050 Series Micro-Computer

assembly language, thus allowing direct access to the features of the Motorola 6809 micro-processor. The typical system configuration is shown in Fig. 17 and provides the micro-processor, memory, keyboard, and connections to a standard television, cassette recorder, RS-232-C port, and joystick inputs as well as the ROM cartridge slot. In this system the ROM cartridge is intended to customize the operation of the machine. Radio Shack and other independent vendors offer a wide range of cartridges including several games, interactive graphics editor, financial packages, text (or word) processor, cassette filing system, and "smart" terminal package. More importantly, one of the independent vendors [5] will take any BASIC program, translate it, and store the result on a ROM cartridge. The cost ranges from $42 to $84 depending upon program size. Using this service, any high level programmer can create a custom processor through firmware.

Conclusions

This report has presented a thorough overview of the impact of read-only memory (ROM) chips on computing procedures and the relation of this impact with other VLSI technologies. It has been demonstrated that ROM chips can be successfully used to store both data and programs. For the numerical analyst, the following are important points:

1. ROM technology is presently available to provide

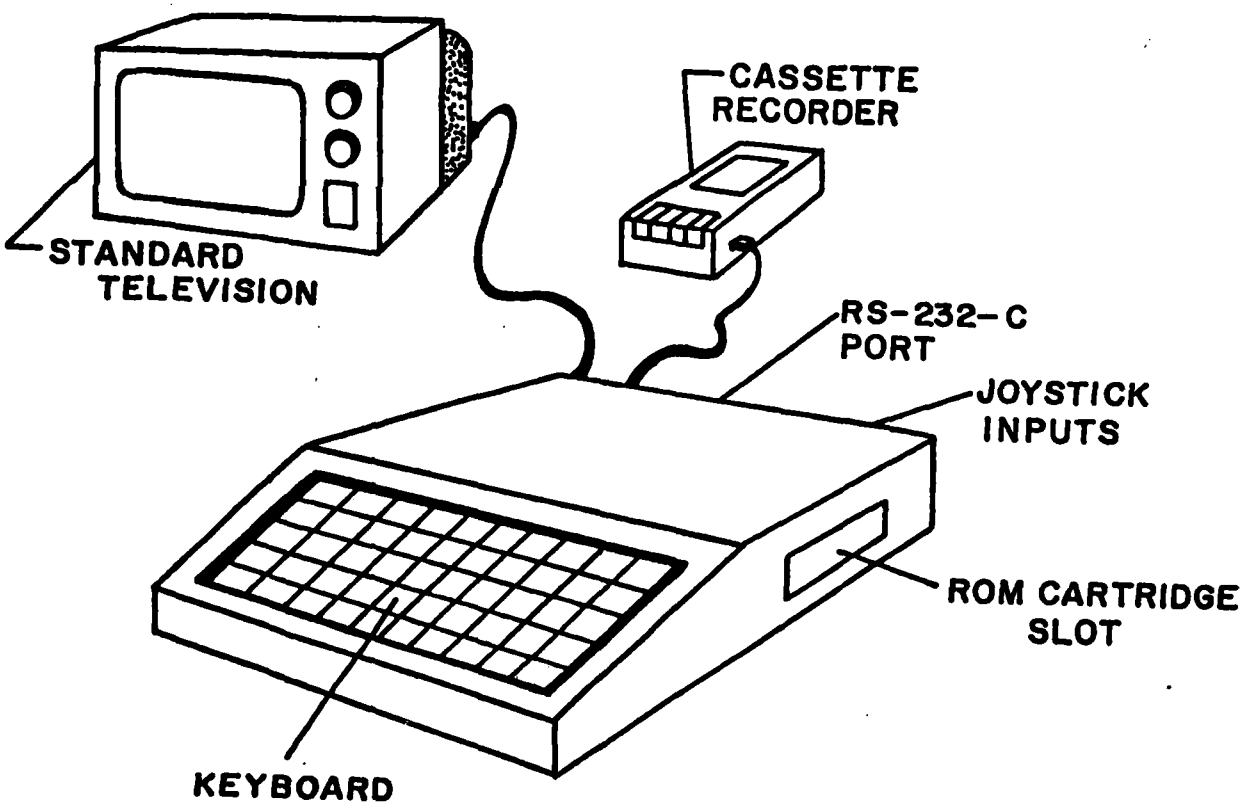


Fig. 17 - Radio Shack TRS-80 Color Computer Configuration.

numerical analysis requirements on a single board.

2. Single chip ROM implementations will be available shortly.
3. Data storage on ROM (or hardware look-up) will provide substantial increases in speed.
4. Both discrete and continuous data may be stored on ROM chips due to the discrete nature of binary encoding.
5. Programs on ROM (or firmware) will allow simplified development of custom processors.
6. Subroutines on ROM (firmware) will reduce the programming cost by eliminating the rewriting of standard computational procedures.
7. Neither micro-code programming nor operating systems on ROM are within the current practical limitations of the numerical analyst, but are predicted to become available as software tools improve.
8. No significant changes are required in programming to accomodate either hardware look-up or firmware.
9. Down-line loading will provide main-frame capabilities on micro and mini-computers
10. ROM applications, including user defined ROM, is beginning to emerge for micro-computers.

Acknowledements

The funding for this research was provided by the Office of Naval Research under contract No. N00014-80-C-0712. This is the second technical report produced under that contract.

References

1. Feeser, L.J., Rooney, M.F., and Shephard, M.S., VLSI Technologies and Numerical Analysis, Technical Report to Office of Naval Research, Contract No. N00014-80-C-0712, Department of Civil Engineering, Rensselaer Polytechnic Institute, Troy, New York, 12181, June 1981.
2. Marks, M., "Impact of VLSI on System Architecture", Scientific American Seminar, Rensselaer Polytechnic Institute, Troy, New York, November 5, 1981.
3. Tektronix Corporation, Beaverton, Oregon.
4. Radio Shack, A division of the Tandy Corportion, Fort Worth, Texas.
5. Eigen Systems, P.O. Box 10234, Austin, Texas.
6. Rooney, Martin F., "Computer Hardware for Civil Engineers", Journal of the Technical Councils, American Society of Civil Engineers, New York, Vol. 107, No. TC1, April, 1981, pp. 153-168.

INDEX

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A110593	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) The Implications of VLSI ROM Chips on Numerical Analysis	5. TYPE OF REPORT & PERIOD COVERED Interim Technical Report
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) L.J. Feeser, M.F. Rooney, M.S. Shephard	8. CONTRACT OR GRANT NUMBER(s) N0014-80-C-0712
9. PERFORMING ORGANIZATION NAME AND ADDRESS Rensselear Polytechnic Institute, Department of Civil Engineering, Troy, New York 12181	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Director Structural Mechanics, Material Science Office of Naval Research, 800 No. Quincy Street, Arlington, VA 22217	12. REPORT DATE ~~Dec. 10, 1981~~ JAN. 8, 1982
	13. NUMBER OF PAGES 57
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document has been approved for public release and sale; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computers, Numerical Analysis, Very Large Scale Integration, Integrated Circuits, Computer Aided Design, Read Only Memory, Firmware, Hardware Look-Up

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report presents a detailed overview of the impact of read-only memory (ROM) chips on computing, and more specifically, numerical analysis. A review of the impact of VLSI technologies is given to indicate the role of ROM chips. A review of available ROM technologies is presented including ROM, PROM, and EPROM chips. A discussion of data storage on ROM is presented covering the mechanics of ROM usage, potential speed increases, programming interfaces, and preparation of data ROM chips through down-line loading. Software on ROM, known as firmware, is discussed similarly and two commercial examples of

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE

20. ABSTRACT (Continued)

firmware are presented. Final conclusions highlight the importance of ROM chips for the numerical analyst.

END

DATE
FILMED

3-82

DTIC