A CROSS-COMPILER FOR THE CYBER 72/M6800 SYSTEM

BY
NAVTEJ S. KHANDPUR

A Project Submitted to the Department of Elecrical Engineering, Ahmadu Bello University Zaria, Nigeria, in Partial Fulfillment of the Requirements for the Award of the Degree of Master of Engineering in Electronics & Telecommunications

June, 1980

The White Rabbit put on his spectacles.
'Where shall I begin, please your Majesty?' he asked.
'Begin at the beginning,' the King said gravely,
'and go on till you come to the end: then stop.'

LEWIS CARROLL
Alice's Adventures in Wonderland

Abstract

This thesis is a report on the design of a high-level programming language for microcomputers, and its associated cross-compiler. The cross-compiler is resident on the University mainframe, and was developed for an M6800 microcomputer.

Chapter 1 discusses microprocessors and microcomputers, to establish the appropriate background and to show the need for a high-level language for microcomputer software development. Programming languages are discussed in Chapter 2, and the language MULANG I is described. In Chapter 3, the structure of a cross-compiler is presented and its workings described. Chapter 4 discusses the implementation of a cross-compiler for MULANG I, while Chapter 5 contains some conclusions and suggestions for further work. An Appendix describes the target M6800 microcomputer system.

ACKNOWLEDGEMENTS

I am exceedingly grateful to the following individuals:

- Dr. B. J. Olufeagba, my Supervisor, who initiated the idea, and lent me his invaluable experience at all times during the execution of the project;
- Mr. P. F. U. Taylor, who imparted much useful advice during informal discussions;
- Mr. M. L. Audu, who painstakingly traced almost all the diagrams;
- Mr. J. O. Ajayi, who typed the manuscript with care and skill.

Thank you all.

N. S. K.

June, 1980.

CONTENTS

1. INTRODUCTION TO MICROCOMPUTERS .. 1.0

1.1. Microprocessors & Microcomputers 1.1

1.2. Microprocessor Evaluation .. 1.9

1.3. Microcomputer System Development 1.14

1.4. Justification 1.20

2. PROGRAMMING LANGUAGES 2.0

2.1. Definitions of Formalisms .. 2.1

2.2. Language Design Concepts .. 2.11

2.3. The M6800 Instruction Set .. 2.18

2.4. The Language MULANG I .. 2.27

3. CROSS-COMPILERS 3.0

3.1. Compilers and Translators .. 3.1

3.2. The Structure of a Cross-Compiler 3.4

3.3. The Lexical Analyser .. 3.10

3.4. The Syntax Analyser .. 3.16

3.5. The Machine Dependent Phases .. 3.20

4. IMPLEMENTATION 4.0

4.1. Introduction 4.1

4.2. The Machine Independent Phases .. 4.6

4.3. Translation of Arithmetic Expressions 4.14

4.4. The Machine Dependent Phases .. 4.22

5. CONCLUSIONS 5.0

5.1. Conclusions 5.1

APPENDIX A - THE M6800 PROTOTYPING SYSTEM A.1

1. INTRODUCTION TO MICROCOMPUTERS

You find sometimes that a Thing which seemed very Thingish inside you is quite different when it gets out into the open and has other people looking at it.

A. A. MILNE

Winnie-the-Pooh

1.1. Microprocessors & Microcomputers

The latest development in computer technology is the microprocessor, a device that has all the functions of the central processing unit (CPU) of a computer. While there is no standard definition to cover the several different varieties of microprocessors, the essential attributes common to all are the use of large scale integration (LSI) circuitry for the CPU and the ability to process both arithmetic and logical data in a bit parallel fashion under the program control.(63) A microprocessor, together with a memory and input/output (I/O) channels for external communications, forms a full-fledged computer called a microcomputer.

Microcomputers are distinguished from minicomputers by their use of LSI, with its attendant lower cost, and from other LSI devices by their programmable behaviour. They have filled a gap in the spectrum of control and processing devices that begins with simple LSI controllers, such as UARTs (Universal Asynchronous Receiver Transmitters), and ranges through fixed function calculators, programmable calculators, and minicomputers.

There is a spectrum of performance within the microcomputer class itself, and a degree of performance overlap with both calculators and minicomputers.

At the low performance end, there is some performance overlap with the more powerful programmable calculators. At the high performance end, there are no functional differences between microcomputers and minicomputers, which employ different (and more expensive) fabrication techniques, which are faster than the LSI used in microprocessor fabrication.

The overall structure of a microcomputer may be represented by three sections: a control section that executes instructions and processes data, a memory section that stores data and instructions, and an input/output section that handles communications between the computer and the outside world. Signals travel from one section to another along paths called buses. Figure 1.1. shows a typical microcomputer. A system clock is provided for timing purposes.

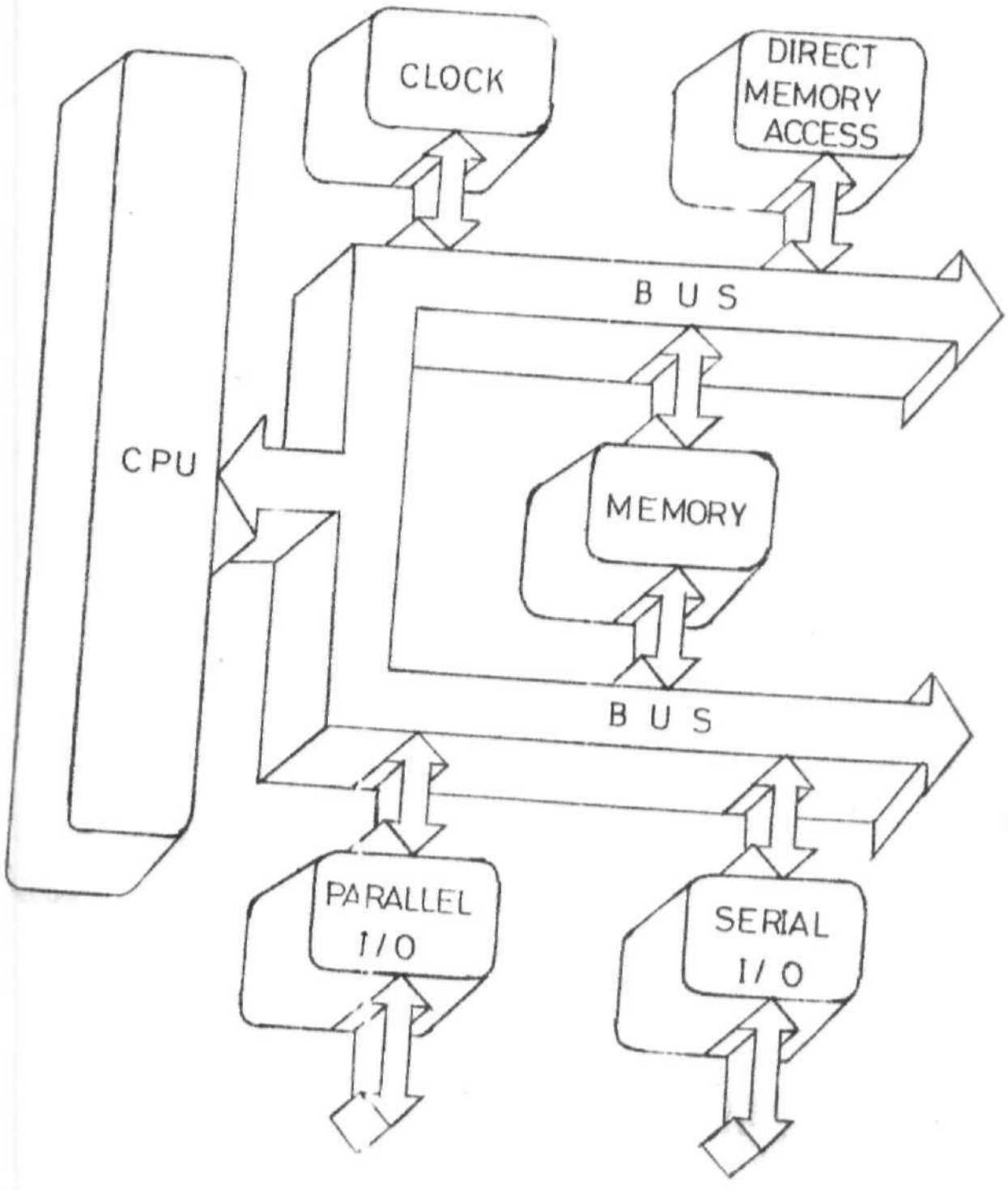


Figure 1.1 - Typical Microcomputer

The control section of the microcomputer is the microprocessor unit. It accepts instructions and performs operations on data. Instructions are fetched from memory, decoded, and then executed. Each instruction specifies a particular cycle of processor operations, known as a microprogram. The working of the microcomputer as a system consists of a succession of these instruction cycles, which are stored on a Read Only Memory (ROM) which may be on-chip or separate. During each instruction cycle, the microprocessor performs many tasks. It generates timing and control signals, performs various tests, transfers data to or from the memory and input/output sections, performs arithmetic and logic functions, and recognizes external signals. Figure 1.2. shows a typical control section.

The keys to the architecture of most CPUs are the registers. Registers are small memories that are part of the control section. The number of registers is normally quite small, however. Data may be saved in a register until a bus or other unit is ready to receive it or until the program requires it. Registers

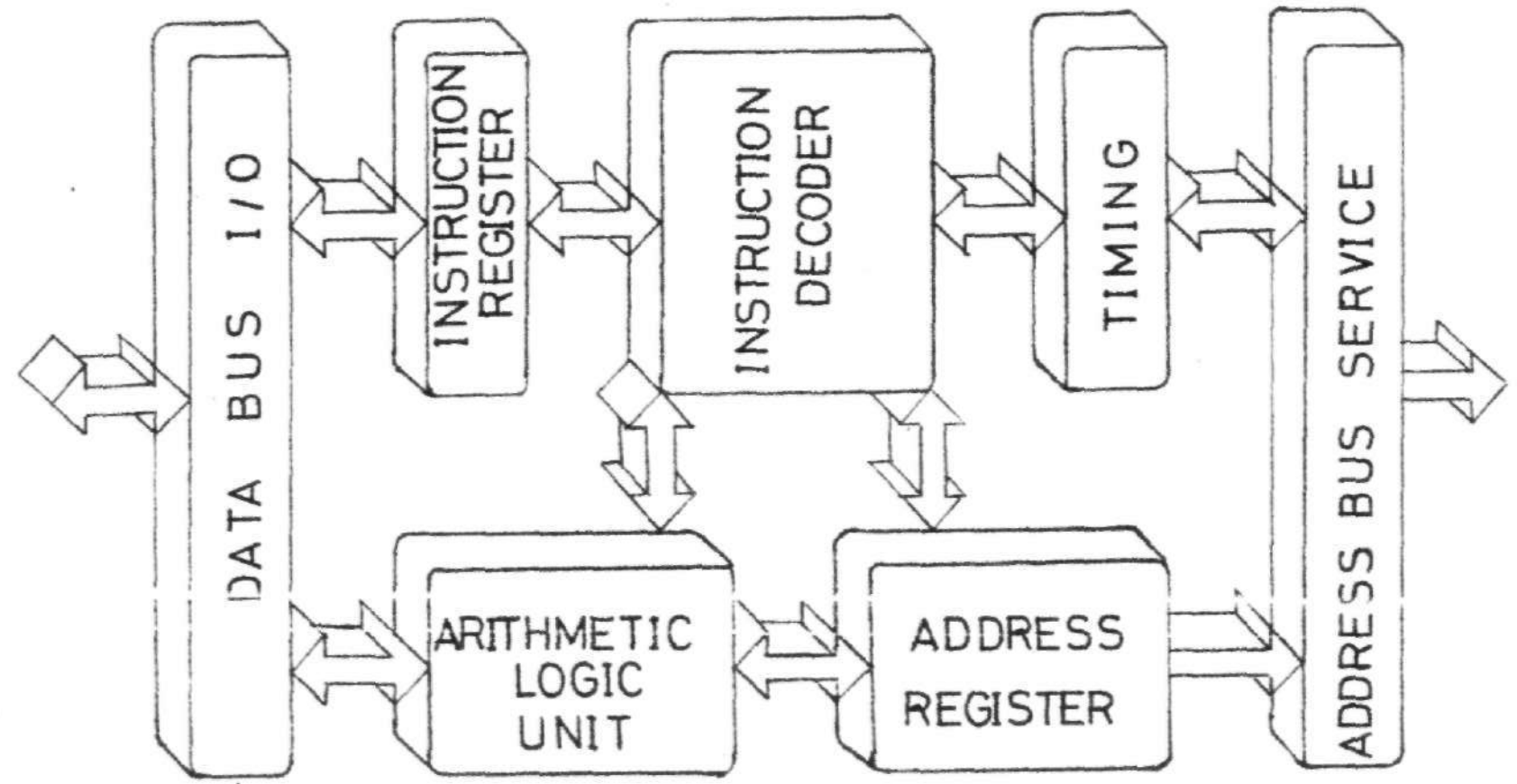


Figure 1,2 - Typical Control Section (M6800)

under program control are advantageous, since the CPU can obtain data from them without a memory access. Registers that are not under program control allow the control section to save data for later use. Figure 1.3. shows a typical set of registers.(44)

The arithmetic and logic unit (ALU) of a microprocessor can vary from a simple adder to a complex unit that can perform many arithmetic and logical functions. If the arithmetic unit cannot perform a function directly, several instructions will be necessary to produce the desired result. For example, in adding two 16 bit numbers in an 8 bit accumulator, several instructions must be executed. These instructions may be microprogrammed on a ROM, and are available to the microprocessor on call. When such an addition is required, the microprogram is decoded and executed.

The microprocessor translates the instruction it obtains from memory (via its registers) into the control signals that produce the desired actions.

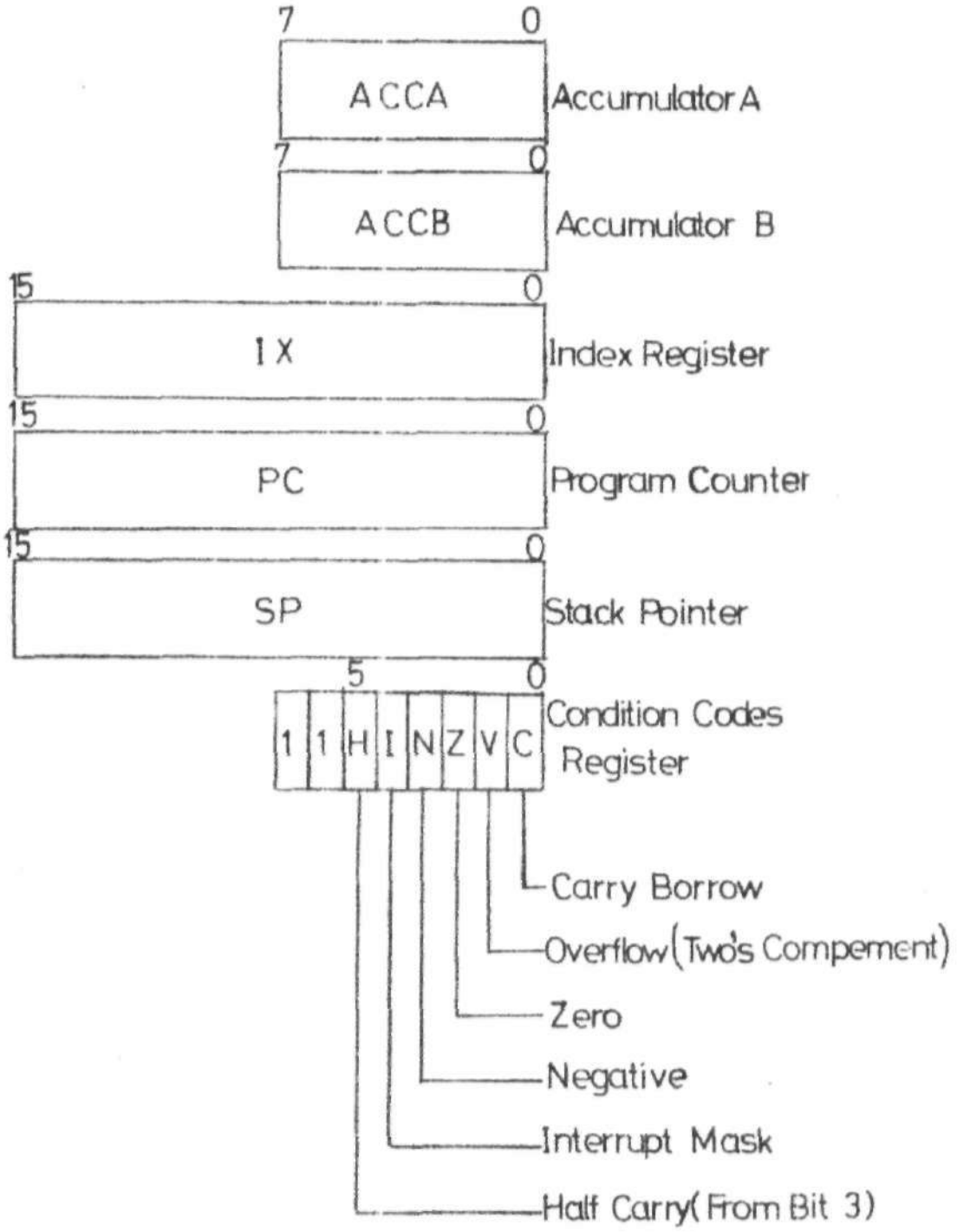


Figure 1,3 - Typical sets of Registers (M6800)

This is performed by the instruction decoder. The control unit of the microprocessor finds addresses for memory locations and registers that the instruction uses, provides function inputs for the ALU and controls the buses so that the instruction is executed properly. The control unit achieves this by supervising the function of the instruction and address handling areas.

The memory section of a microcomputer contains storage units, which usually consist of semi-conductor cells. The memory is organized into the shortest groupings of bits that the computer can handle at a time or into words, which have the same bit length as the computer's data registers, data buses, and arithmetic unit. The memory is arranged sequentially and each word has a unique address. The address decoder on receipt of an address sent by the control unit, selects the proper memory location. The amount of time taken to find the correct memory location and to obtain its contents is called the access time of the memory.

There are many different memory addressing techniques available on different microprocessors. Among the most widely used include:

* Direct
* Indirect
* Immediate
* Indexed
* Relative

Microcomputers use many variations and combinations of these techniques. A comprehensive survey of these techniques may be found in References 40, 59 and 64.

The input/output section of a microcomputer handles the transfer of data between the computer and external devices or peripherals. The transfer involves status and control signals as well as data. The input/output section reconciles timing differences between the computer and the peripherals, formats the data properly, handles status and control signals, and supplies the required current and voltage levels. Irregular transfers may be handled with interrupts, control signals that receive the immediate attention of the control

section and cause the suspension of its normal activity.

There are three techniques currently in use to service peripherals(59)

* Polling
* Interrupts
* Direct Memory Access.

In polling, the CPU sequentially checks each peripheral to see if it needs servicing. If no servicing is required, the next peripheral is interrogated. If, however, the peripheral requires servicing, the CPU refers to the corresponding service routine before moving on to the next peripheral. At the end of polling, the CPU returns to its previous task or repeats the cycle. The disadvantage of this system is its slow speed - much time may be wasted in polling peripherals which do not require servicing.

The interrupt scheme is more practical in real-time situations because it is asynchronous. Each time a device needs servicing, it attracts the attention of

the CPU, which then completes execution of the present instruction, stores the contents of all internal working registers, identifies the calling device, and services it. After servicing, the CPU continues with the execution of the next instruction. Most microprocessors cater for several interrupts, and some even distinguish between device priorities.

When bulk high-speed transfers of data between memory and peripherals are necessary, the microprocessors' speed limitations preclude its involvement in such transactions. Modern microcomputers have a direct link between the memory and the input/output section that allows the transfer of data to or from peripherals without intervention by the control section. This method of transferring data is direct memory access (DMA). The advantage of DMA is that the transfer speed is limited only by the memory access time. Data transfers through the control section require several instruction cycles and take 10 to 20 times as long as DMA. However, DMA causes the microprocessor to go into a 'wait' state, and so DMA can take place

for periods less than the refresh time of the memory. Most DMA schemes involve 'cycle-stealing' - making use of clock cycles when the microprocessor is idle.

1.2. Microprocessor Evaluation

There are a number of features to consider when evaluating a microprocessor. The more important features are listed in Figure 1.4.(63)

* Data Word Length
* Programming Flexibility
  -----Instruction Set
  -----Addressing Modes
* Interface Flexibility
* Choice of Memories
* Interrupt Feature
* Additional Packages
* Speed
* Architecture

Figure 1.4. - Features to Consider when Evaluating Microprocessors.

1.2.1. Word Size

The number of bits that are processed concurrently is referred to as the data word length of the microprocessor. Common word lengths are 1, 2, 4, 8, 12 and 16 bits. In general, the longer word lengths provide higher system throughput, and more powerful memory addressing, while the shorter word lengths require somewhat less hardware for the CPU. Instructions that operate on data of fractional and multiple word lengths allow wider applications. Microprocessors which allow operations on fractions of words are known as bit-slice microprocessors. Bit-slices may be joined together to form words of different sizes. The saving in execution time is obvious, since only bits that need to be operated upon are processed. Bit-slice technologies and programming are complex, and are considered beyond the scope of this thesis.

1.2.2. Instruction Set

The instruction set affects the application flexibility, system throughput, memory program storage requirements, and ease of programming. The memory addressing modes constitute one of the more important considerations for most applications. Direct, relative, indexed and

indirect addressing modes are all important for program efficiency. Because the power of individual instructions and methods of counting may vary widely, the number of instructions in the set executed by a microprocessor is a poor index of its usefullness.(63) The only realistic method of comparing instruction sets is to code benchmark programs typical of the intended applications and compare the execution times and number of bits of storage required, but even this is not conclusive.

1.2.3. Interface Structures

The interface structure of a microprocessor for general purpose applications should be easily adaptable to a variety of hardware organizations without a high overhead in hardware or software. The application may demand anything from a sophisticated, high-speed, bidirectional, autonomous bus with address and data sharing the same lines (multiplexing), to a simple low cost bus having separate input, output and address lines, and whose control is heavily dependent on the processor. The microprocessor input/output circuitry should directly interface with the tripolar logic

required to derive input/output lines.

1.2.4. Memory Considerations

The choice of memories for a given microprocessor is important since the memory is often a major portion of the system cost. Read/write memories are commonly used for variable data storage and for program storage during program development. Program storage for prototype or preproduction systems is often in programmable read-only memories (PROMs) while read-only memories (ROMs) are used during high-volume production.

1.2.5. Interrupt Features

Hardware interrupt features are essential to many microprocessor applications and can sometimes simplify the software. As a minimum, the interrupt should cause the microprocessor to automatically save the current contents of the program counter and branch to an interrupt service routine. Provision for several levels of nested interrupts is desirable. This capability may be provided by a last-in, first-out stack memory, onto which the content of the program counter is pushed for successive nested interrupts. In order to return from the interrupts, the stack is then 'popped', causing

entries to be retrieved in the reverse order of entry. The stack is also used for storing the program counter contents during subroutine calls and for storing temporary data. The stack may be implemented as a pointer to external read/write memory, which allows a very large stack, or as hardware internal to the microprocessor, which offers high speed and the important advantage of using it in systems implemented with read only memory.

1.2.6. Other Considerations

The number of packages required in addition to the basic LSI processor is important to less complex applications in which the microprocessor is a significant part of the total system cost. Areas likely to require external components are clock signal generators and system timing, memory and input/output control, memory and input/output buffering, and interrupt control.

The different measures of execution time, including static time, cycle time and time to add two numbers are rather meaningless as indicators of how rapidly a processor can perform a particular task when taken by

themselves. The only way to make this determination is to actually code the particular task, or a task with similar requirements.

Also important is the functional architecture of a microprocessor. Number of general registers, input/output organization and convenience of transferring data and preserving status must all be considered.

1.3. Microcomputer System Development

The wide range of products available and the many different configurations possible make it difficult to come up with quantitative guidelines for microprocessor applications. However, some very general comments can be made.

1.3.1. Microprocessor Applications

Most microprocessor applications require from 15 to 40 integrated circuits (IC's) for the complete processor - including clocks, control logic and memory and peripheral buffering, but exclusive of memory storage and peripheral interface IC's (which may add from 5 to 50

or more components).[63] In most practical cases, it is realistic to consider microprocessors for any sequential digital designs requiring more than 50 or 60 IC's using hard-wired logic, having more than a trivial number of steps to the flowchart and having some logical or arithmetic data processing requirements. Speed requirements must be relatively low although in those cases where high throughput is required, it may be worthwhile to investigate the use of multiple microprocessors.

Microprocessors find applications in many areas, and some typical areas are:

* Point of sale terminals
* Word Processors
* Inventory control systems
* Process, numeric, and machine control
* Automatic test systems
* Digital instruments
* Traffic controllers
* Communications systems
* Navigational systems

* Consumer goods.

As people come to understand the computer as a processor of strings, lists, stacks, and queues and as an intelligent control device, strictly numeric oriented applications will continue to recede into the background. The capabilities of microcomputers are particularly well suited to meet this growing awareness.(27)

1.3.2. Hardware Tools

Before a microprocessor system is used, it must be checked thoroughly. Prototyping hardware is essential to develop and debug the programs and hardware for the end product. The prototyping hardware should provide expanded memory capability for storage of debug programs, a teletypewriter interface for printout, a paper or cassette tape interface for program loading, and a control panel that provides access for examination and alteration of all registers and memory locations and single instruction stepping.

1.3.3. Software Tools

Software support is required for developing and debugging application programs. Software tools are the special-purpose computer programs which the software

engineer can use to facilitate the development of the product software.(61)

It is possible to develop product software without using software tools. However, it is rarely practical to do this. In fact, the quality, cost, and development time cycle of the entire product is directly related to the appropriateness of the software tools used to develop the software. This is especially true since in most microcomputer applications, software development accounts for 50 - 80 percent of total development costs.(35, 61)

There are five main categories of software tools: (a) editors, (b) translators (assemblers and compilers), (c) loaders, (d) simulators, and (e) debuggers. Each of these tool categories facilitates some stage of the program development process. Editor programs facilitate the creation of the source program. Translator programs create the object program from the source program (see Chapter 3). Loader programs transfer the object program from some external medium, such as a

paper tape, into the microcomputer memory. Simulator programs make possible a substantial portion of the testing of an object program when the microcomputer is unavailable. Debugger programs facilitate the testing of an object program on the microcomputer.

All of these categories of tools are divided into two classes: (a) cross-computer tools and (b) resident tools. Cross-computer tools are programs which run on some computer other than the microcomputer for which software is being developed. The availability of such cross-computer tools makes possible the simultaneous development of microcomputer hardware and software, since software development is not dependent on the availability of the product or the prototyping hardware.

The other class of software tools are resident programs. These are programs which run on the microprocessor for which software is being developed. Generally, these tools are available for purchase from the microprocessor manufacturers as part of a total hardware-software

development system. In order to run these programs, the system developer must either purchase or develop a microcomputer prototyping system.

The tradeoffs between using cross-computer versus resident tools can be summarized as follows:

(a) cross-computers are generally faster machines than the microprocessors for which they produce programs;

(b) the initial costs to use the cross-computer tools are usually lower;

(c) the overall costs to use cross-computer tools are usually higher;

(d) the cross-computer method is generally more convenient, since the peripherals are generally faster, and if the cross-computer is a timesharing system, several programmers can be using the system at one time;

(e) the cross-computer method requires that the programmer(s) learn two computer systems versus one for the resident tool approach.

In a University or other situations where computers charges are very low, and where most programmers are familiar with the Resident mainframe computer, trade-offs c and e are not applicable. Thus cross-computer tools are cheaper and more feasible to use than resident tools.

1.4. Justification

Selecting a microprocessor for a system design is difficult, for the choice can mean the difference between success and failure, profit and loss. However, many of the criteria involved are not readily apparent. Engineering considerations, such as selection of speed and word length are relatively simple to handle. But there are other factors that do not show up on the manufacturers' data sheets yet considerably considerably influence costs for better or worse. Consider, for example, startup costs, where decisions such as selection of the level of the language in which programming will be done and choice of a development system will affect the cost of programming.(35)

As well as affecting startup costs, the choice of a programming language influences the speed of evaluation and the amount of memory needed. The question here is which level of language to look at. There are essentially three choices: assembly language, macro-assembly language, and high-level language.

Assembly language, the simplest available, is a one-for-one symbolic equivalent for machine code. A major advantage is that it simplifies address calculations. With an assembler, an experienced programmer will produce a 30% more efficient code than a novice,(35) as Figure 1.5. suggests. Memory-use efficiency in the figure is defined as the number of bytes of memory required for a program in software language, with 100% efficiency assigned to a program in assembly language created by an 'experienced' programmer bent on saving memory space.

There is one important difference between assembly language and macro-assembly language: the latter can define strings of instructions that can be called by

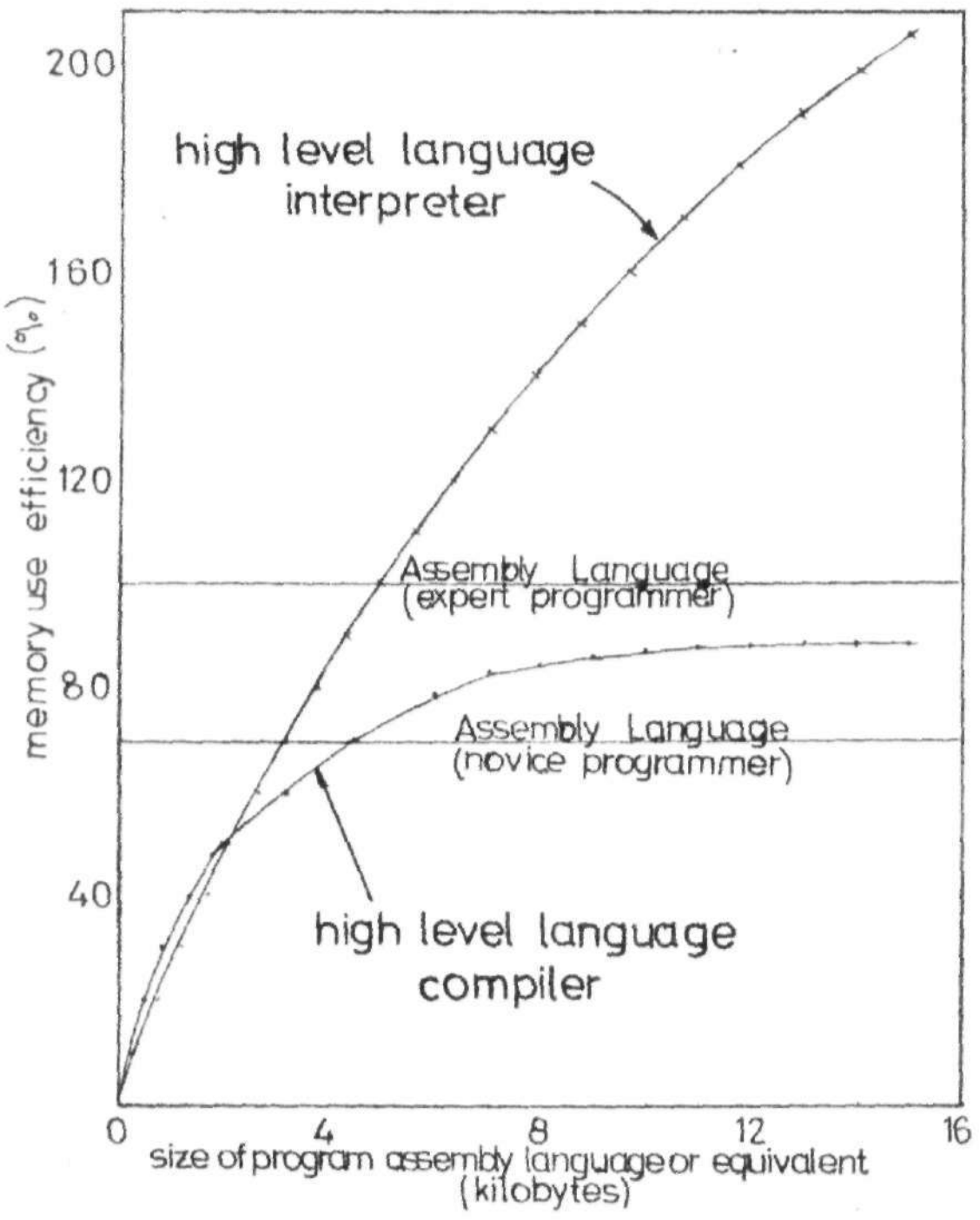


Figure 1,5 – Microprocessor Memory-Use Efficiencies (35)

just one symbol. This conveniently shortens coding and eliminates errors. The savings in programming costs over assembly language will run between 5% and 20% depending on the experience of the programmer.(35) However, selecting a macro-assembler has no effect on evaluation speed or efficiency memory use.

A high-level language will permit the programmer to code in terms related to a natural language and mathematical expressions. The source statements that a programmer writes can be translated into machine code and specialized subroutines added. The outcome is a package compiled from several routines and machine language statements.

The compiler and the size of the program determine the relative efficiency of this programming method. For small programs, efficiency is relatively low, because the specialized subroutines are used only once or twice. Figure 1.6. shows that, as a program grows, memory-use efficiency can reach 90%.(35) It may be said that assembly language is only slightly more efficient

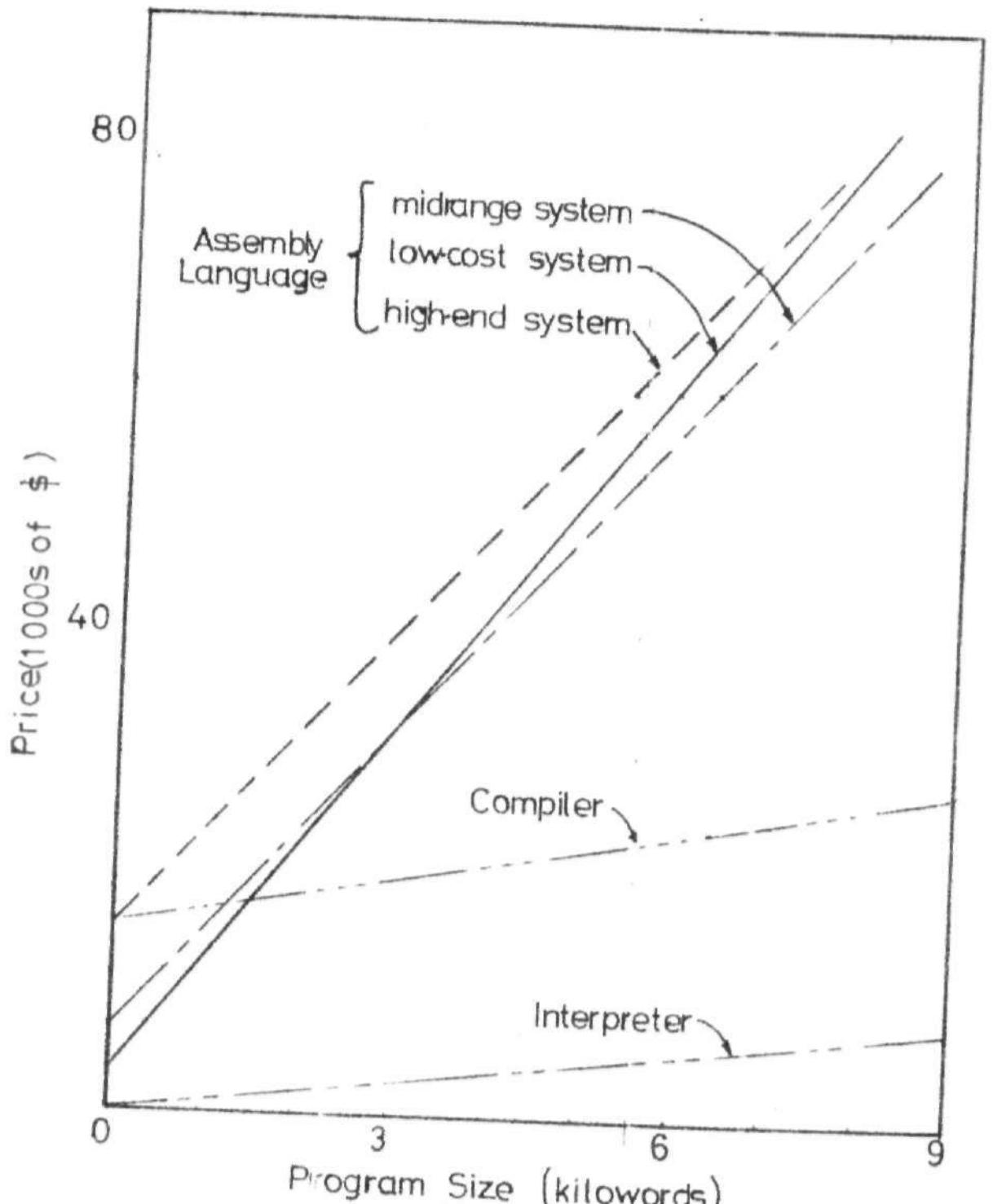


Figure 1,6 - Microcomputer Programming Costs

than a high-level language in terms of memory use.

Numerous studies[38, 66, 67] in the past few years have confirmed that the average computer applications programmer can only write 10 -15 debugged program statements per day, regardless of the programming language he uses. Bearing in mind that one high level language statement corresponds to ten or more assembly language statements, we see that it is faster, and hence cheaper, to write programs in a high-level language.

In this thesis, a high-level language was designed, and a cross compiler to interface between the University Computer Centre's Cyber 72 mainframe and the Electrical Engineering Department's M6800 prototyping system, was constructed for the language. Subsequent chapters are on language design, cross compilers, and implementation.

2. PROGRAMMING LANGUAGES

After great pain, a formal feeling comes.

EMILY DICKNSON

"341"

2.1. Definitions of Formalisms

A formal system is an uninterpreted calculus or logistic system.[21] It consists of an alphabet, a set of words called axioms, and a finite set of relations called rules of inference. A formal system is uninterpreted in the sense that no meaning is formally attached to the symbols of the system; there is for each of the above-mentioned systems a standard informal interpretation of the symbols, but other interpretations are generally valid so far as the systems themselves are concerned.

Formal systems are generally constructed in order to have formalized models of informal, intuitive notions. A formal model can be studied mathematically and if the model is appropriate, the results may indicate much about the notions that it portrays.

Formal systems are used to define the syntax of a programming language. Such definition is important both to the user and to the implementer of the language. The user needs for reference a clear description of the

language, while the implementer is faced with the problems of transferability and maintenance. If the same language is to be implemented on different machines, the legal strings of the language must be well-defined so that the user-level appearance is, so far as is practicable, invariant. Furthermore, the implementer must be concerned with maintenance. Both the user and the maintainer of a compiler need an exact specification of the acceptable strings of the corresponding language.

2.1.1. Language Description Systems

Formalisms such as the context-free grammar or syntax-directed translation serve as good tools for language description. The context-free grammar (Backus-Naur or Backus-Normal Form) is an excellent description for the syntax of several languages. Syntax-directed translation, in its full generality, is a good vehicle for defining the translation to be performed by the compiler when the syntax is specified by a context-free grammar.

Certain classes of the theory of context-free languages serve to identify maximal classes of grammars with desirable properties, such as the existence of a simple parsing algorithm. Furthermore, language theory can warn the compiler writer that certain programs caannot be written.(1)

By providing a general mathematical framework for describing many compiler processes, language theory helps the compiler designer to express many important design and optimization problems.(1)

A language may be intuitively thought of as a set of sentences with well defined structures and meaning. The rules specifying valid constructions of a language are its syntax: the syntax of a language describes its forms. The assignment of a meaning, or interpretation, to symbols is the semantics of a language.

If an algorithm exists that will successively produce legal sentences, such a listing algorithm is called a generative specification of the language. If the algorithm generates sentences in such an order that each new sentence has at least as many symbols as the

previous sentence, it is clearly possible to decide whether or not a given string is in the language. If such a decision can be made in finite time for every string, the language is called decidable.(21)

An alternative type of algorithm could be used to specify languages. This algorithm accepts as input the string to be tested, analyzes the input, performs whatever computation is necessary,and produces an answer indicating whether or not the string is a legal sentence of the language. This is called an analytic specification. A language with an analytic specification is decidable if the analyzer always stops after finitely many steps for every input string.(21)

2.1.2. Definitions of Language Description Systems

It is necessary to develop a formalism in which language definitions can be stated. This defining language is the syntactic meta-language, while the language being talked about is the object language. A formal system is a meta-language. Some definitions (after Donovan(21)) are presented below, to establish a universe of discourse.

An alphabet T is a finite set of symbols. A sentence is a concatenation of symbols. For any set U, U* represents the set of all possible concatenations of elements of the set U. λ represents the null or empty string - the string which contains no elements.

A language, L, is a subset of the set of finite concatenations of symbols in an alphabet T. This is written $L \subset T^*$. The terminal symbols are the symbols of the alphabet T. The non-terminal symbols are a set of N of symbols not in T that represent intermediate states in the generation process. The starting symbol is a distinguished non-terminal symbol from which all strings of the language are derived.

A production is a string transformation rule having a left hand side that is a pattern to match a substring of the string to be transformed, and a right hand side that indicates a replacement for the matched portion of the string. It is important to note that any substring of the current string may be replaced by an applicable production and that only that part

of the string matched by the left hand side is affected. Productions can totally replace substrings, or they may merely rearrange the symbols of the matched substring.

A formal grammar is a finite set of productions, which contain both terminal and non-terminal symbols. One of the non-terminal symbols is the starting symbol of the grammar. Mathematically,

$G = (N, T, \Sigma, P)$ where

N is the set of non-terminal symbols

T is the set of terminal symbols

Σ is the starting symbol; $\Sigma \in N$

P is the set of productions $\alpha \rightarrow \beta$ where α, $\beta \in (NUT)^*$, $\alpha \neq \lambda$ and $N \cap T$ is empty.

If a string α may be changed into a string β by a single application of a single production then β is directly derived from α, written

$$\alpha \Rightarrow \beta$$

Similarly, a derivation is a sequence of strings

$$\alpha_0, \alpha_1, \alpha_2, \ldots\ldots\ldots, \alpha_n$$

such that

$$\alpha_{i-1} \Rightarrow \alpha_i , \quad 0 < i <= n$$

A sentential form is any string which can be derived from the starting symbol Σ, while a sentence is a sentential form containing only terminal symbols.

Finally, a language L defined by a grammar G $(L(G))$ is the set of sentences that can be derived from Σ in G. L is called ambiguous if it contains a sentence for which there is more than one distinct proof in which the left-most non-terminal is replaced at each step. The definition of productions allows for a wide variety of string transformations. In the Chomsky hierarchy,[33,43,47] the most general form of grammar is type 0, in which the form of the productions is unrestricted, while grammar types 1, 2 and 3 are categorized by increasing restrictions on the form of the productions.

A type 0 (zero) or contracting context-sensitive gram-

mar contains productions of the form

$$\alpha \rightarrow \beta$$

where α is a non-empty string of non-terminal symbols, and β is a string of terminal and/or non-terminal symbols.

A type 1 or noncontracting context-sensitive grammar contains only productions of the form

$$\alpha \rightarrow \beta$$

where α is a non-empty string of non-terminal symbols, β is a non-empty string of terminal and/or non-terminal symbols and the length of β is not less than the length of α .

A type 2 or context-free grammar contains only productions of the form

$$\alpha \rightarrow \beta$$

where α is a single non-terminal symbol and β is a string of terminal and/or non-terminal symbols.

A type 3 or regular expression grammar contains only

productions which are one of the forms

$$\alpha \longrightarrow a$$
$$\alpha \longrightarrow a\beta$$
$$\alpha \longrightarrow \beta a$$

in which α and β are single non-terminal symbols, a is a single terminal symbol.

It may be noted from the definitions that type 2 grammars define the syntax of statements, declarations, expressions and so on while type 3 grammars define the micro-syntax of lexical items such as names, numbers and strings. To determine the type of a grammar, it is necessary to look at the left hand and right hand sides of every production. If every left hand side is a single symbol the grammar is type 2 or type 3; then if every right hand side is one of the type 3 forms it is a type 3 grammar, otherwise type 2. If any of the productions in the grammar has more than one symbol on its left hand side then the grammar is type 0 or type 1: then if every production is in the restricted type 1 format it is a type 1 grammar, other wise it is type 0.

The grammar types form a hierarchy because type 3 is a restricted form of types 0, 1 and 2, type 2 is a restricted form of types 0 and 1 and type 1 is a restricted form of type 0. Corresponding to the hierarchy of grammars is a hierarchy of languages. Some languages are simple enough to be described by a type 3 grammar, some are too complex for type 3 grammars and require a type 2 grammar, and so on.

2.1.3. The Backus - Naur Form

The Backus-Naur Form - also called the Backus Normal Form - BNF is a meta-language for writing grammars that is commonly used to specify the syntax of programming languages. In BNF non-terminals are written as names enclosed in corner brackets '< >'. The sign $\longrightarrow$ is written : : = (read 'is replaced by'). Alternative ways of rewriting a given non-terminal are separated by a vertical bar, | , (read ('or'). The BNF meta-language is equivalent to context free grammar, so class symbols expand without reference to surrounding context. To expand with reference to context a more powerful formal system is needed, one with the capability of cross-reference between elements

of the sentence structure that it generates. Cross reference is a context-sensitive feature, and more powerful methods for defining languages are canonic systems - which are beyond the scope of this thesis.

2.2. Language Design Concepts

There are a number of basic concepts underlying any computer language design. These concepts include selection by name (the idea of a set), selection by position (the idea of an ordered set), subdefinition in terms of components, iteration, meaningful primitive operations, and control over the allocation of computer resources.(43)

Many features may be provided in a language, but the addition of features to a language is not a linear process as far as the translator is concerned. Constructs interact with each other, and the addition of a single feature may, in some cases, double the size of the translator. The potentially exponential growth of translator size as language complexity increases

has two important implications. First, a translation method effective for simple test languages or machines may prove inadequate for more practical languages or existing machines. Second, an elaborate language may need a much larger translator than needed by several smaller languages which, in aggregate, have the same features.

2.2.1. Language Applications

The first consideration in the design of any language is the proposed applications of that language.(43) The application of the language will largely determine the operators, data structures and flow of control features provided in that language. For example, a language which is to be used almost exclusively for character data manipulation should provide for extensive string handling operations such as concatenation, linked list data structures, indexing, etc., whereas a language designed principally for scientific work has little or no need for some of those features, but requires provisions for various arithmetic operations, floating point numbers, library functions, etc. There is a trend now to design general-purpose programming

languages, which could be used for a wide variety of applications. However, no truly general purpose language has been successfully implemented to date.

2.2.2. Data Types and Structures

The next consideration in the design of a language are the data types and structures to be provided. Data means relevant values and their associated internal representations. A language needs the facility to represent data explicitly (constants), to assign their representations to memory locations and later to recover them (variables), and to examine and combine them (operations). Four kinds of data are particularly useful: bits, for recording logical operations; strings of bits, for packing information; integers, for all kinds of arithmetic; and strings of characters, to represent text. The data types which can be provided will depend to an extent on the machine on which the object code will be executed. The word size and instruction set are factors to consider in the specification of data types. This is especially important in microcomputers where word sizes are small and instruction sets limited.

Data types may be enhanced into data structures such as ordered sets of values - arrays, stacks, tables, pointer spaces, queries, lists and the like. Their implementation involves a compromise between the language complexity and the machine capability.

Efficiency demands that the constructs proposed be close to the machine. This matching must occur in several areas. Memory access must be efficient by operating on the directly addressable quanta of memory. This is very important in microprocessors, where direct addressing is the fastest mode of memory access. Furthermore, the language must be concise enough to produce object programs short enought to fit into the generally small memory of microcomputers. Another factor for consideration is the interaction between the translator and the target machine. In resident compilers, the operating system takes control of the execution phase, but in cross-compilers, the mode of transfer of object code must be explicitly considered.

In a program, it is necessary to specify not only what operations are to be performed but also their order, sequences and operands (data). Adoption of a convention - such as execution in order of appearance in the program - frees the designer to indicate explicitly only the exceptional cases.

A basic concept is selection from an ordered set. For ease of expression and efficiency of operation, both data and instruction are usually arranged so that the appropriate element can be conveniently selected from a set of alternatives. Similarly, it is possible to select among alternative instruction sequences depending on the value of an expression. Commonly, the expression is logical (two-valued), and the set of instruction sequences contains two elements. The usual representation for this selection is the IF e_1 THEN e_2 ELSE e_3 ENDIF conditional statement, where e_2 and e_3 are the alternative instruction sequences, and e_1 is the logical expression used for the selection.

Repetition of selected sequences of operations is another basic requirement of control. Two forms of repetition are particularly important: repeating the operation a specified number of times - often with specific values for a controlled variable, and repeating them only as long as a particular condition is satisfied.

Transfer of control also takes two forms. We may either temporarily transfer to a subprogram (possibly supplying it with some parameters) which will return to the original instruction sequence, or we may simply enter the instruction sequence at some other point, with no provision for return.

2.16. Program Maintainability

Most programs, but especially production programs, need to be maintained, modified and studied by people, who may not be the original authors. Many programs are read more by people than by computers. The language therefore should be conducive to producing readable program statements.

The reader of a program should be able to easily follow

the logic of the program at two levels. First, he should be able to quickly determine the major structural units, their intent, and the flow of control among them. Second, he should be able to determine precisely what operations are being performed at any given point in a program.

It is important that it be possible to break the program down into both logical (block structured) and visual (paragraphed) units. Enough freedom should be allowed in the choice of names attached to various program entities (labels, variables, etc.) to make them mnemonic and easy to understand. Provision must be made for comments to clarify the intent of the subprogram.

To be readable, a language for the computation of complex processes must permit these processes to be defined in terms of less complex processes, which can then be defined in terms of even simpler processes... and ultimately in terms of the basic operations of the language. There is a necessity for hierarchical

program structures based on these basic operations.

2.3. The M6800 Instruction Set

An instruction is one of the binary inputs to which the Central Processing Unit reacts with a specified series of actions.(40) Each instruction must specify an operation, the sources of the data, the destination of the result, and the address of the next instruction. In microcomputers, must of this instruction is implicit - that is independent of the particular instruction. This use of implicit information results in shorter instructions but means that more instructions are necessary in order to perform common tasks. The standard format is the one-address instruction in which the computer obtains one operand from an accumulator, places the result back in the accumulator, and sequences through a program by incrementing the program counter after each step.

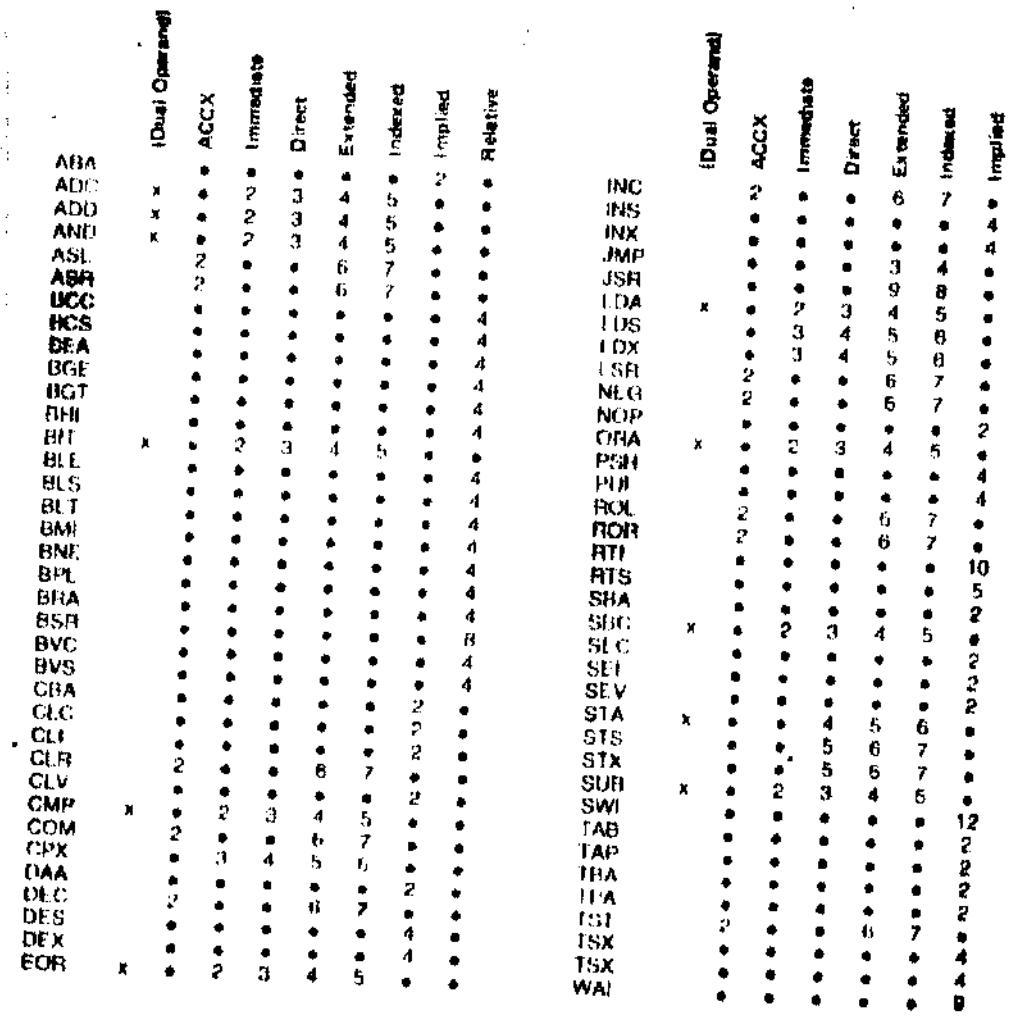
Many different methods can be used to specify the address of the data. Direct addressing is simple and

can handle single data items. Indirect and indexed addressing are more complex but allow the program to modify the actual or effective address so as to handle arrays and tables. Immediate addressing is useful for constants, whereas relative addressing allows the transfer of entire programs from one place in memory to another. Stack addressing depends on the last-in, first-out order of the stack; it is short and fast but difficult to use. Register addressing methods allow the programmer to place data or addresses and use them repeatedly without additional memory accesses.

The instruction set itself can be divided into several basic categories: (a) data manipulation instructions that actually change the data, such as arithmetic, logical, shift, comparison, or special-purpose operations; (b) data movement instructions that move the data from one place to another, such as memory transfers, register transfers, input/output operations, and stack transfers; (c) program manipulation instructions that alter the normal sequence of operations, such as unconditional and conditional jumps, subroutining

operations, halts, and no operations; (d) status management instructions that affect only the machine status, such as enable and disable interrupts.(40)

The M6800 instruction set contains 72 separate executable instructions, which assemble into 1 to 3 bytes (8 bits) of machine code.(44) The number of bytes depends on the particular instruction and on the addressing mode.The instruction set, together with the addressing modes which are available for use with the various instructions are indicated in Figure 2.1. The coding of the first (or only) byte, corresponding to an executable instruction, is sufficient to identify the instruction and the addressing mode. The hexadecimal equivalents of the binary codes, which result from the translation of the 72 instructions, in all valid modes of addressing, are shown in Figure 2.2. There are 197 valid machine codes, 59 of the 256 possible codes being unassigned. When an instruction translates into two or three bytes of code, the second byte, or the second and third bytes, contain(s) an operand, an address, or information from which an address is obtained during execution.(44)



Note: Interrupt time is 12 cycles from the end of the instruction being executed, except following a WAI instruction. Then it is 4 cycles.

Figure 2.1 - M6800 Instruction Set, Addressing Modes and Associated Execution Times (44)
(Times in Machine Cycles)

LSB / MSB	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	*	NOP (INH)	*	*	*	*	TAP (INH)	TPA (INH)	INX (INH)	DEX (INH)	CLV (INH)	SEV (INH)	CLC (INH)	SEC (INH)	CLI (INH)	SEI (INH)
1	SBA	CBA	*	*	*	*	TAB (INH)	TBA (INH)	*	DAA (INH)	*	ABA (INH)	*	*	*	*
2	BRA (REL)	*	BHI (REL)	BLS (REL)	BCC (REL)	BCS (REL)	BNE (REL)	BEQ (REL)	BVC (REL)	BVS (REL)	BPL (REL)	BMI (REL)	BGE (REL)	BLT (REL)	BGT (REL)	BLE (REL)
3	TSX (INH)	INS (INH)	PUL A (INH)	PUL B (INH)	DES (INH)	TXS (INH)	PSH A (INH)	PSH B (INH)	*	*	RTS (INH)	*	RTI (INH)	*	*	SWI (INH)
4	NEG (A)	*	*	COM (A)	LSR (A)	*	ROR (A)	ASR (A)	ASL (A)	ROL (A)	DEC (A)	*	INC (A)	TST (A)	*	CLR (A)
5	NEG (B)	*	*	COM (B)	LSR (B)	*	ROR (B)	ASR (B)	ASL (B)	ROL (B)	DEC (B)	*	INC (B)	TST (B)	*	CLR (B)
6	NEG (IND)	*	*	COM (IND)	LSR (IND)	*	ROR (IND)	ASR (IND)	ASL (IND)	ROL (IND)	DEC (IND)	*	INC (IND)	TST (IND)	JMP (IND)	CLR (IND)
7	NEG (EXT)	*	*	COM (EXT)	LSR (EXT)	*	ROR (EXT)	ASR (EXT)	ASL (EXT)	ROL (EXT)	DEC (EXT)	*	INC (EXT)	TST (EXT)	JMP (EXT)	CLR (EXT)
8	SUB (A) (IMM)	CMP (A) (IMM)	SBC (A) (IMM)	*	AND (A) (IMM)	BIT (A) (IMM)	LDA (A) (IMM)	*	EOR (A) (IMM)	ADC (A) (IMM)	ORA (A) (IMM)	ADD (A) (IMM)	CPX (IMM)	BSR (REL)	LDS (IMM)	*
9	SUB (A) (DIR)	CMP (A) (DIR)	SBC (A) (DIR)	*	AND (A) (DIR)	BIT (A) (DIR)	LDA (A) (DIR)	STA (A) (DIR)	EOR (A) (DIR)	ADC (A) (DIR)	ORA (A) (DIR)	ADD (A) (DIR)	CPX (DIR)	*	LDS (DIR)	STS (DIR)
A	SUB (A) (IND)	CMP (A) (IND)	SBC (A) (IND)	*	AND (A) (IND)	BIT (A) (IND)	LDA (A) (IND)	STA (A) (IND)	EOR (A) (IND)	ADC (A) (IND)	ORA (A) (IND)	ADD (A) (IND)	CPX (IND)	JSR (IND)	LDS (IND)	STS (IND)
B	SUB (A) (EXT)	CMP (A) (EXT)	SBC (A) (EXT)	*	AND (A) (EXT)	BIT (A) (EXT)	LDA (A) (EXT)	STA (A) (EXT)	EOR (A) (EXT)	ADC (A) (EXT)	ORA (A) (EXT)	ADD (A) (EXT)	CPX (EXT)	JSR (EXT)	LDS (EXT)	STS (EXT)
C	SUB (B) (IMM)	CMP (B) (IMM)	SBC (B) (IMM)	*	AND (B) (IMM)	BIT (B) (IMM)	LDA (B) (IMM)	*	EOR (B) (IMM)	ADC (B) (IMM)	ORA (B) (IMM)	ADD (B) (IMM)	*	*	LDX (IMM)	*
D	SUB (B) (DIR)	CMP (B) (DIR)	SBC (B) (DIR)	*	AND (B) (DIR)	BIT (B) (DIR)	LDA (B) (DIR)	STA (B) (DIR)	EOR (B) (DIR)	ADC (B) (DIR)	ORA (B) (DIR)	ADD (B) (DIR)	*	*	LDX (B) (DIR)	STX (B) (DIR)
E	SUB (B) (IND)	CMP (B) (IND)	SBC (B) (IND)	*	AND (B) (IND)	BIT (B) (IND)	LDA (B) (IND)	STA (B) (IND)	EOR (B) (IND)	ADC (B) (IND)	ORA (B) (IND)	ADD (B) (IND)	*	*	LDX (IND)	STX (IND)
F	SUB (B) (EXT)	CMP (B) (EXT)	SBC (B) (EXT)	*	AND (B) (EXT)	BIT (B) (EXT)	LDA (B) (EXT)	STA (B) (EXT)	EOR (B) (EXT)	ADC (B) (EXT)	ORA (B) (EXT)	ADD (B) (EXT)	*	*	LDX (EXT)	STX (EXT)

DIR = Direct Addressing Mode
EXT = Extended Addressing Mode
IMM = Immediate Addressing Mode
IND = Index Addressing Mode
INH = Inherent Addressing Mode
REL = Relative Addressing Mode
A = Accumulator A
B = Accumulator B
*Unimplemented Op Code

Figure 2.2 - M6800 Instruction Map (Hexadecimal)(44)

The data manipulation instructions available on the Motorola 6800 are:

(i) Arithmetic Operations

ADD or ABA	Add
ADC	Add with carry
DEC	Decrement
DES or DEX	Double decrement
INC	Increment
INS or INX	Double increment
SUB or SBA	Subtract
SBC	Subtract with carry

(ii) Logical Operations

AND	And
COM	Ones complement
NEG	Twos complement
EOR	Exclusive Or
ORA	Or

(iii) Shift Operations

ASR	Arithmetic shift right
ASL	Logical shift left
LSR	Logical shift right
ROL	Left rotate with carry
ROR	Right rotate with carry

(iv) Comparison Operations

BIT	Bit test
CMP or CBA	Compare
CPX	Compare double
TST	Test

The data transfer instructions available on the Motorola 6800 are:

(i) Memory Transfer Operations

LDA	Load
LDS, LDX	Load double word
STA	Store
STS	Store double word (stack pointer)
STX	Store double word (index register)

(ii) Input/Output Operations

The Motorola 6800 has no specialized input/output instructions. Each input/output device is associated with an input/output channel management device that is associated with a memory address, so that any instruction that transfers data to or from

memory can perform input or output. The most common such instructions are LDA and STA (Load and Store) respectively. Other instructions may have less obvious effects, however. For example, transfer of a zero to an output device may be achieved by clearing the associated memory address. Input data can be examined by using CMP, CBA, CPX or TST (compare or test) instructions with the input device as the address.

(iii) Stack Operations

PSH	Push data on stack
PUL	Pull data from stack

(iv) Internal Transfer Operations

TAB	Transfer accumulators A → B
TAP	Transfer accumulators to Condition Code Register
TBA	Transfer accumulators B → A

TPA	Transfer Condition Code Register to accumulator
TSX	Transfer stack pointer to Index Register
TXS	Transfer Index Register to stack pointer

The program manipulation instructions available on the Motorola 6800 are:

(i) Unconditional Jump Operations

BRA	Branch always
JMD	Jump

(ii) Conditional Jump Operations

BCC	Branch if carry clear
BCS	Branch if carry set
BEQ	Branch if equal to zero
BGE	Branch if greater than or equal to zero

BGT	Branch if greater than zero
BHI	Branch if higher
BLE	Branch if less or equal
BLS	Branch if lower or same
BLT	Branch if less than zero
BMI	Branch if minus
BNE	Branch if not equal to zero
BPL	Branch if plus
BVC	Branch if overflow clear
BVS	Branch if overflow set

(iii) Subroutining Operations

BSR	Branch to subroutine
JSR	Jump to subroutine
RTS	Return from subroutine.

(iv) Halts and No Operations

NOP	No operation
WAI	Wait for interrupt.

The only status management instructions available on the M6800 are those that enable and disable the interrupt system.

CLI	Clear and enable interrupt
SEI	Set and disable interrupt
RTI	Return from interrupt
SWI	Software interrupt.

It can be seen that the functions of some instructions overlap with those of other instructions. However, an attempt shall be made to use as many different instructions as possible, depending on the status of the machine at a given time.

2.4. The Language MULANG I

MULANG I is the source language developed in this thesis to be used with the associated cross-compiler. The language is first considered from the viewpoints of applications, data types, target machine, control structure and readability.

Initially, it is expected that programs written in the language MULANG I shall execute simple small-integer arithmetic and logical operations. To this end, facilities for integer arithmetic and logical expressions have been provided in the language. In the prototype, no immediate need is envisaged for text-processing, and as such no character string handling facilities are provided. The same applies to fixed and floating point numbers, and their associated arithmetic. Later, versions of the language will cater for these needs.

The data types allowed in MULANG I are INTEGER and LOGICAL. As mentioned above, the expected applications

of the language will require only integer and logical operations. Therefore, only those data types have been provided for. Provision has also been made for one dimensional arrays (vectors) of both data types. The data type set, as well as the data structures may be included in future versions of MULANG I as and when the need arises.

The target machine for the language is the Motorola M6800 microprocessor. This machine has an 8 bit data word, and a 16 bit address word. Thus, one word integer data may take the values 0 to 255 or -127 to 128 depending on the implementation, and the machine may address up to 65,536 bytes of memory directly. The instruction set of the microprocessor has been discussed in the previous section. The target system is described in detail in Appendix A. The mode of transfer of the object code from the host machine to the target machine will be manual. This is because no common input/output medium for the two machines is presently available.

2.4.3. Program Readability

The structured constructs available in MULANG I are the IF, DO and WHILE constructs. The IF construct allows conditional execution of expression sequences, based on a relational test. The DO and WHILE constructs allow repetition of expression sequences, for the DO construct a fixed number of times, and for the WHILE construct until a relational condition is safisfied. The three constructs permit multiple nesting, so that considerable program flexibility may be achieved. The constructs are discussed separately later.

MULANG I has been written in such a way that programs written in MULANG I can be made very readable. Programs can be written in a modular fashion, and paragraphing may be used to improve readability. Any number of comments may be interspersed between any two statements in the program. A comment is regarded as any character string which follows a '?' in column 1 of a line (similar to comments in FORTRAN). Considerable freedom has been allowed in the choice of variable names, which may be up to 10 alphanumeric characters long, subject to two conditions: variable

names must begin with alphabetic characters; and that a variable name must not be a reserved word (terminal symbol) such as DO, IF, PROGRAM, INTEGER, etc. The notation used in arithmetic statements is the common infix notation (e.g. 2 + 3) rather than the post-fix notation - also known as the reverse Polish notation - (e.g. 2 3 +). The infix notation was preferred for reasons of readability and ease of use, even though it is more difficult to implement than the post-fix notation. A formal description of MULANG I is presented in Figure 2.3. The BNF meta-language is used for the description. The structure of a typical program written in MULANG I is examined, and each statement is considered in turn.

2.4.4. MULANG Program Structure

Every program written in MULANG I must comprise a main program, and, optionally, one or more subprograms. The main program must precede the subprograms, if any. This is because all variables, labels, input/output channels and constants must be declared in the main program. Apart from this declarative part, and the program header statement, the main

Note: the following symbols are meta-symbols belonging to the BNF meta-language, and are not symbols of the language MULANG.

::= | { }

The curly brackets denote possible repetition of the enclosed symbols zero or more times. In general,

A ::= {B}

is a short form for the purely recursive rule

A ::= <empty>|AB

<totprog> ::= <program><totend>

<program> ::= <mainprogram> {<subprogram>}

<totend> ::= END PROGRAM;

<mainprogram> ::= <proghead><block><progend>

<subprogram> ::= <subhead><stmprt><subend>

<proghead> ::= PROGRAM <idn>;

<subhead> ::= SUBPROGRAM <idn>;

<block> ::= <dclprt><stmprt>

<dclprt> ::= <vartype><info>

<vartype> ::= <inttype><logtype>

<inttype> ::= <INTEGER><vadist>;

<logtype> ::= LOGICAL <varlist>;

<info> ::= <labels><channels><constants>

<labels> ::= LABELS <idn>{, <idn>};

Figure 2.3. BNF Description of MULANG I

```
<channels>    ::= CHANNELS <idn> {, <idn>};
<constants>   ::= CONSTANTS <lit> {, <lit>};
<stmprt>      ::= <simpstm>|<strucstm>|<simpstm><stmprt>|
                  <strucstm><stmprt>
<strucstm>    ::= <ifstm>|<dostm>|<whilestm>
<ifstm>       ::= If(<e1>)THEN; <e2> ELSE; <e3> ENDIF;
<dostm>       ::= <begindo><stmprt><enddo>
<begindo>     ::= DO <idn> = <index>;
<index>       ::= <op2>,<op2>,<op2>
<enddo>       ::= ENDDO;
<whilestm>    ::= <condition><stmprt><endwhile>
<condition>   ::= WHILE(<e1>)PERFORM;
<endwhile>    ::= ENDWHILE;
<e1>          ::= <op1><relop><op2>
<e2>          ::= <stmprt>
<e3>          ::= <stmprt>|<blank>
<op1>         ::= <idn>|<arrayel>
<op2>         ::= <op1>|<lit>
<simpstm>     ::= <assnstm>|<callstm>|<jumpstm>|<iostm>|
                  <label>:<simpstm>
<assnstm>     ::= <idn> = <expr>;|<arrayel> = <expr>;
<expr>        ::= <lit>|<idn>|<arrayel>|<expr><op><expr>|(<expr>)
<callstm>     ::= CALL <idn>;
<jumpstm>     ::= JUMPTO <label>;
```

Figure 2.3. BNF Description of MULANG I (Continued)

```
<iostm>        ::= <instm> | <outstm>
<instm>        ::=  INPUT( <channel> )  iolist   ;
<outstm>       ::=  OUTPUT(<channel>)  iolist  ;
<channel>      ::= <idn>
<iolist>       ::= <iostrt> , <iolist>
<iostrt >      ::= <var> | <var> ( <lit> ) | <var> (<idn> )
<progend>      ::=  END <idn> ;
<subend>       ::=  RETURN <idn> ; | <label> : <subend>
<varlist>      ::= <var> | <var> ( <lit> )
<var >         ::= <idn> | <idn> {, <idn>}
<blank>        ::=  <    >
<arrayel>      ::= <idn> ( <lit> ) | <idn> ( <idn> )
<label>        ::= <idn>
<op>           ::=  + | - | * | / | = | **
<relop>        ::=  > | < | ¬ | ⊏ | & | ^
<idn >         ::= <letter> | <letter> <letterdigit>
<letterdigit>  ::= <letter> | <digit>  (1 to 5)
<lit>          ::= <digit>
<letter>       ::=  A | B | ..... | Z
<digit>        ::=  0 | 1 | 2 | .... | 9
```

Figure 2.3. BNF Description of MULANG I (Continued)

program and the subprograms are similar in all respects. They both contain a statements part, where all operations to be performed within the block are stated. These include arithmetic expressions, conditional statements, loops, and subprogram calls. The last statement in a program block is the END statement. The structure of a typical program is shown in Figure 2.4.

Coding lines in MULANG I have an 80 column structure, so as to allow effective use of punch cards for program entry. A coding line may be one of 3 types - a statement, a continued statement, or a comment.

A statement may start from any place on the coding line, and may use some or all of the available spaces on that line. Statements are terminated by a semi-colon (;). Statements which cannot be finished on one coding line may be continued on the next, but only one semi-colon (at the end of the continued line) will terminate the statement. A comment would have a '?' (question mark) in column 1 of the coding line,

```
PROGRAM name  ;

     declarative part

     statementspart

END name  ;
SUBPROGRAM name  ;

statement part

END name ;
   :
   :
   :
SUBPROGRAM name  ;
     statements part
     END name  ;
     END PROGRAM  ;
```

Figure 2.4 - Structure of a Typical MULANG Program

followed by any string located between columns 2 and 80. A comment which cannot be finished on one line may be continued on the next, provided there is a '?' in Column 1 of the next line. Comments are not compiled, and may therefore contain any sequence of alphanumeric or special characters.

The character set used by MULANG I is as follows:

Alphabetic	A - Z	
Numeric	0 - 9	
Special	=	equals
	+	plus
	-	minus
	*	asterisk
	/	solidus
	^	cap
	&	ampersand
	#	number
	>	greater than
	<	less than
	]	right bracket
	[	left bracket

) right parenthesis

(left parenthesis

; semi-colon

: colon

? question mark.

Executable statements of MULANG I may be identified by using alphanumeric identifiers as labels. Statement labels are separated from the rest of the statement by a colon. In a given program, each statement label is global, so that it must uniquely identify a statement. Subscripted variables with one subscript only are allowed. The subscript must refer to a single element in a declared vector. Subscripts must be integers or integer variables.

The set of reserved words (terminal symbols) for MULANG I are classified as:

(a) header words

PROGRAM SUBPROGRAM

(b) declarative words

INTEGER LOGICAL CONSTANTS LABELS

CHANNELS

(c) structured construct words

IF THEN ELSE ENDIF DO ENDDO
WHILE PERFORM ENDWHILE

(d) executable jump words

CALL JUMPTO RETURN

(e) input/output words

INPUT OUTPUT

(f) end word

END

2.4.5. MULANG Statements Description

The statements of MULANG I are now considered in the order in which they may occur.

The header line of a program, which serves as the header line of the main program, is of the form

PROGRAM idn ;

where idn is an alphanumeric identifier which uniquely identifies the program block. The header line of a subprogram has the form

SUBPROGRAM idn ;

where idn identifies the subprogram. This name must be used in all references to the subprogram.

The declarative part, which follows the header line in the main program only, is used to specify variable names and types, constants, labels and input/output channels. Integer and logical variables are declared using the statements

```
INTEGER     list ;
```

and

```
LOGICAL     list ;
```

respectively, where 'list' is the list of identifiers to be assigned the type. Identifiers are separated by commas, and may include one-subscripted identifier. Integer subscripted identifiers declare vectors of that type. Constants are declared using the statement

```
CONSTANTS   list ;
```

where 'list' is the list of integer constants, separated by commas, used in the program. Labels are declared using the statement

```
LABELS      list ;
```

where 'list' is the list of identifiers used as labels in the program. A label may not be a variable as well. Channels for input and output are declared by using

the statement

CHANNELS (no_1 = $address_1$, no_2 = $address_2$,.....,
no_N = $address_N$) ;

where no_i is the channel number and $address_i$ is the physical address of that channel.

The declarative part is useful in implementing the compiler - it simplifies to a large extent the table building routines necessary in the lexical phase. The need for declaration of variable types, constants and labels is clear. The channels statement is provided to allow effective use of the M6800's unique input/output capability. An integer number is associated with an input/output channel address. This address can be referred to during input and output by referring to the channel number. The advantage in using channel numbers is that if they are arranged sequentially, they may be used from inside loops, and if specified by variables, they may reference many different channels with relative ease.

The statement part follows the declarative part in the

main program, and the header statement in subprogram. Statements can be part of control structures, or they can be executable statements. Control structures are three in number: the IF construct, the DO construct, and the WHILE construct.

The IF construct has the general form

```
IF (logical relation) THEN;
     Statements 1
ELSE;
     Statements 2
ENDIF;
```

The logical relation has the form

Op 1 relation Op 2

i.e. it is a binary operation. Op 1 may be a variable, or an array element, and Op 2 may be a variable an array element, or a logical or integer constant. The relation may be any one of

=	equal to
>	greater than
<	less than
]	greater than or equal to

⊏ less than or equal to

& logical AND

∧ logical OR

During execution, the value of the logical relation is computed: if true, statements 1 are executed; if false, statements 2, which may optionally be empty, are executed. The construct is represented diagramatically in Figure 2.5.

The DO constant has the form

```
DO   identifier = indices;
     Statements
ENDDO;
```

The identifier is an integer variable which is used to keep a count. It may be used by the statements which follow the DO construct header statement, for example, as an array pointer. Its value, however, must not be changed by the statements. The indices are a set of three integer values (constants or variables) separated by commas. The first index indicates the initial value for the identifier declared

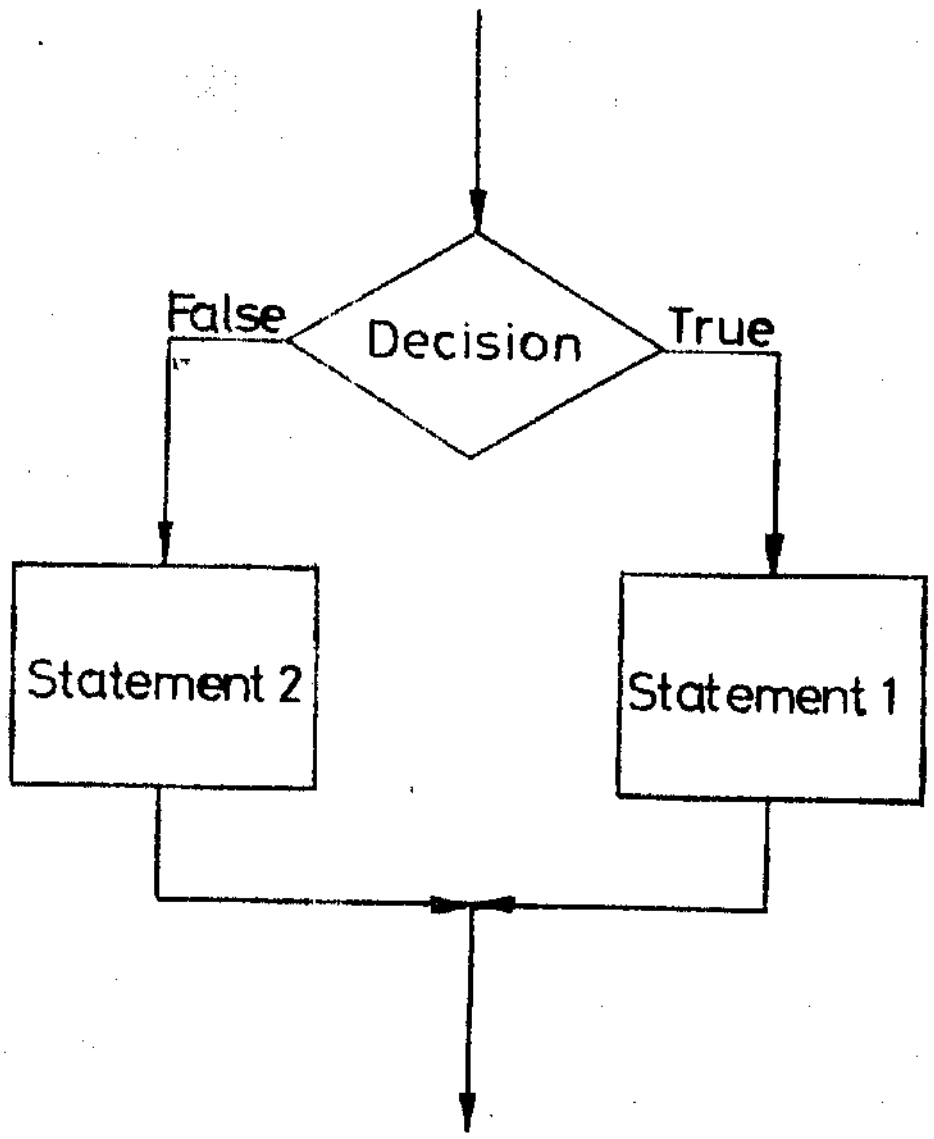


Figure 2.5 - The IF-THEN-ELSE Construct

before the equals sign. The second index indicates the increment to be added to the current value of the identifier to obtain the next value. It may be positive or negative. The third index is the value of the identifier at which the looping in the construct will cease. This value must be greater than the initial value if the increment is positive, and less if the increment is negative. The statements part of the construct is a set of valid executable statements of the language. Thus multiple nesting may be achieved The DO construct ends with an ENDDO. Multiple nesting of DO loops is permitted.

In the DO construct, the statements following the header statement are executed repeatedly and the value of the identifier incremented by index 2 on every loop, until such a time that the identifier value equals or exceeds the termination value, index 3. The construct may be represented diagramatically as in Figure 2.6.

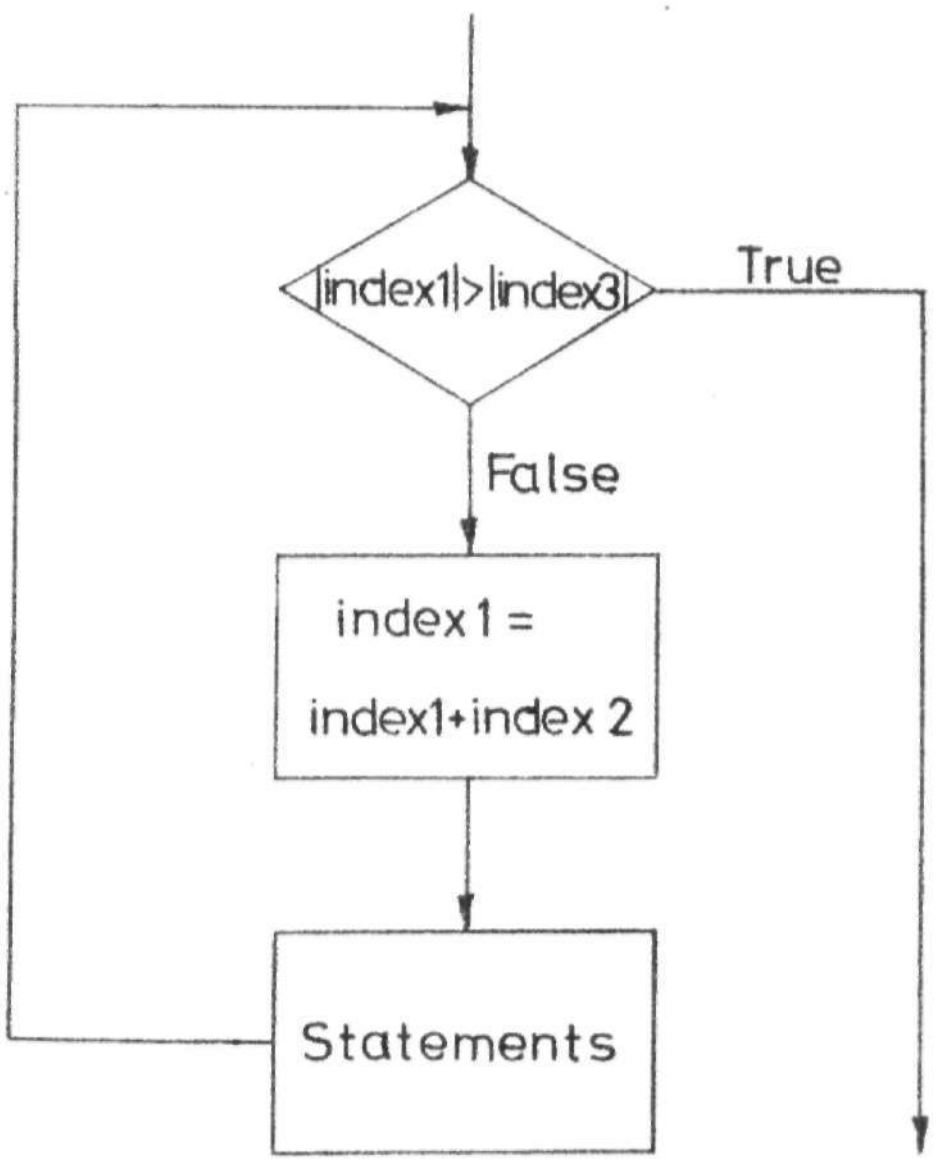


Figure 2.6 -The DO Construct

The WHILE construct has the form

```
WHILE (logical relation) PERFORM;
        Statements
ENDWHILE;
```

where the logical relation and statements are as described in the constructs above.

This statement construct causes repeated execution of the statements in the construct as long as the logical relation holds true. Execution ceases when the logical relation has a value FALSE. Generally, one of the operands in the logical relation is modified during execution of the statements. Otherwise, there is no exit from the construct except by the use of a JUMPTO - which defeats the purpose of the WHILE construct. Figure 2.7 shows the WHILE construct.

No entry may be made <u>into</u> any of the three control structures described above, as this will cause unpredictable behaviour. Exits are allowed from any point in the construct.

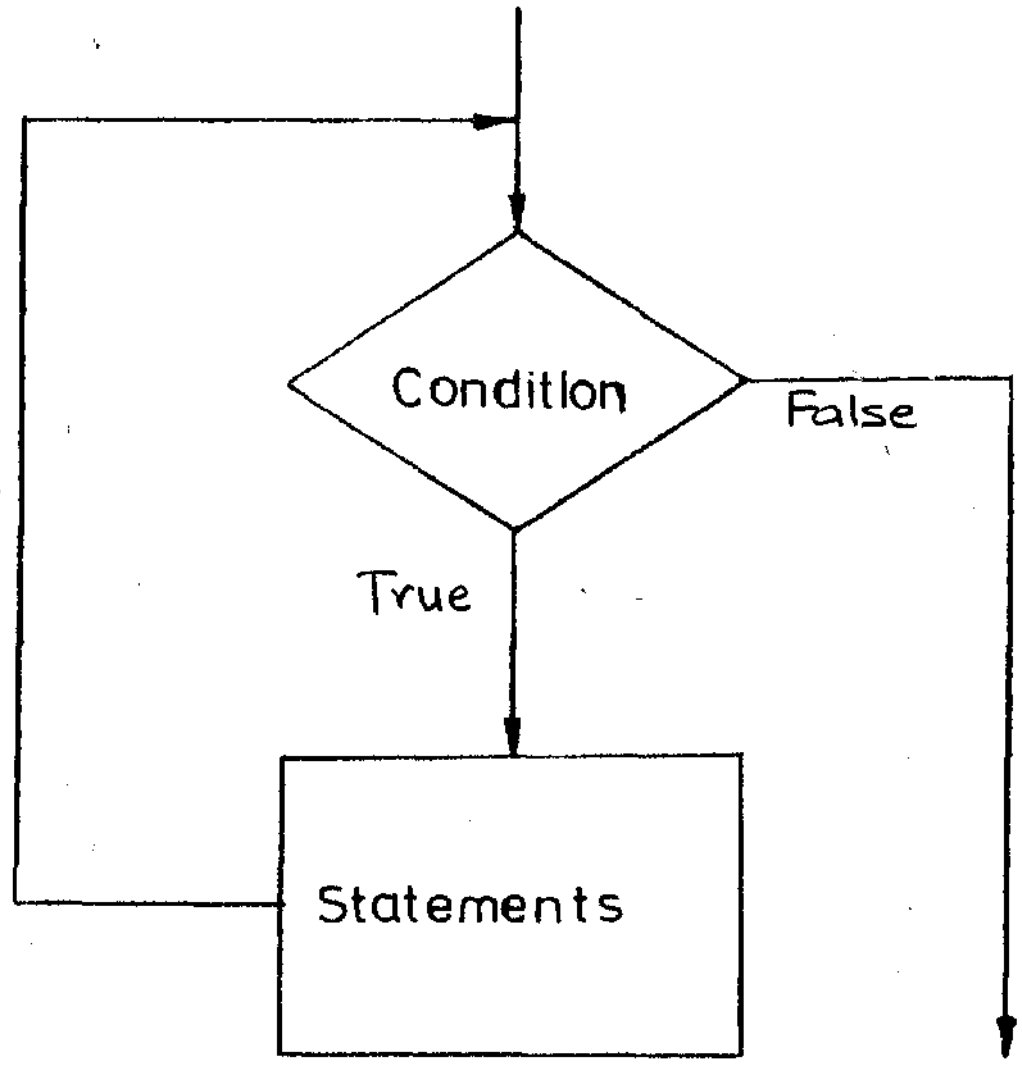


Figure 2,7 - The WHILE Construct

The structured constructs all contain statements which are executable. The executable statements of MULANG I are the assignment statements, the CALL statements, the JUMPTO statements, the INPUT and OUTPUT statements, the RETURN statements, and the END statements.

The assignment statements are replacement statements - in the form of arithmetic expressions, using regular operator precedence. Unary expressions, however, are not allowed, so that negation must be accomplished by subtracting from zero. All operands in an assignment statement are assumed positive.

CALL statements have the form

CALL idn ;

On encountering this statement, the program transfers control to the first executable statement in the subprogram identified by 'idn'. The statements in the subprogram are executed until the first RETURN statement is encountered. Control is then passed to the statement immediately following the CALL statement in the calling program.

RETURN statements have the form

RETURN;

JUMPTO statements have the form

JUMPTO label;

and have the effect of unconditionally transferring control to the statement identified by the label 'label'. No jump may be made out of one program block into another program block. Thus jumps may not be used to transfer control from a calling program to a sub-program.

External communication is effected by the use of the INPUT and OUTPUT statements. These have the form

INPUT (idn) list; and

OUTPUT (idn) list;

where idn is an integer variable whose value identifies a channel, and list is a list of variables to be read (INPUT) or output (OUTPUT). Provision of a channel number allows for INPUT/OUTPUT from various peripherals.

Program blocks are terminated with the END statement,

of the form

END idn;

where idn is the alphanumeric identifier of the program block. Every header statement must be matched by a corresponding END statement.

The entire program-main program and all subprograms must be terminated by an ENDPROGRAM statement, which is of the form

END-PROGRAM;

This indicates to the compiler that the program is complete, and all characters detected after this statement are regarded as garbage.

In summary, the following important characteristics of MULANG I may be listed:

1. Declaration of variables is mandatory.
2. Certain key words are 'reserved' and cannot be used as identifiers.
3. The semi-colon (;) is considered as a statement delimiter, not a statement terminator.
4. The standard data types are those of integer

and logical values. The basic data structuring facility is the vector, with an upper bound less than 5 digits.

5. There is a JUMPTO statement. Labels are alphanumeric identifiers and need to be declared. Labels are separated from statements by a colon (:).
6. The control structures are the DO constructs, the WHILE constructs and the IF-THEN-ELSE-ENDIF constructs.
7. Multiple nesting of these constructs is allowed.
8. Assignment statements follow standard operator precedence relationships.
9. Multiple assignments are not allowed.
10. Recursion is not allowed, i.e. a subprogram may not call itself or its calling program.
11. Subprograms are called by name reference.
12. Input/output statements must specify channels, which need to be declared.
13. All variables are set initially to zero for integers and FALSE for logical.

MULANG I lacks cettain 'favourite features' such as dynamics arrays, automatic type conversions, default declarations, etc. These were not oversights but deliberate omissions. In some cases their presence would be primarily an invitation to inefficient programming solutions - which is not desirable for microcomputer applications. In other cases, it was felt that they would be contrary to the aim of clarity and reliability and generally, to good programming style.

3. CROSS-COMPILERS

'Explain all that,' said the Mock Turtle.
'No, No ! The adventures first,' said the Gryphon in an impatient tone: 'Explanations take such a dreadful time.'

LEWIS CARROLL
Alice's Adventures in Wonderland

3.1. Compilers and Translators

A translator is a software system (written in the implementation language) that transforms statements of one computer language (the source language) into statements in some other computer language (the object or target language).[4] If the source language is a high-level or procedure-oriented language such as FORTRAN or Pascal, and the object language is a low-level language such as an assembly language or machine language, then such a translator is called a compiler.

Executing a program written in a high-level programming language is basically a two-step process, as illustrated in Figure 3.1. The source program must first be compiled, that is, translated into the object program. Then the resulting object program is loaded into memory and executed.

Resident compilers produce object code for the computer on which they are hosted. Thus compilation and execution of programs are both carried out on the host

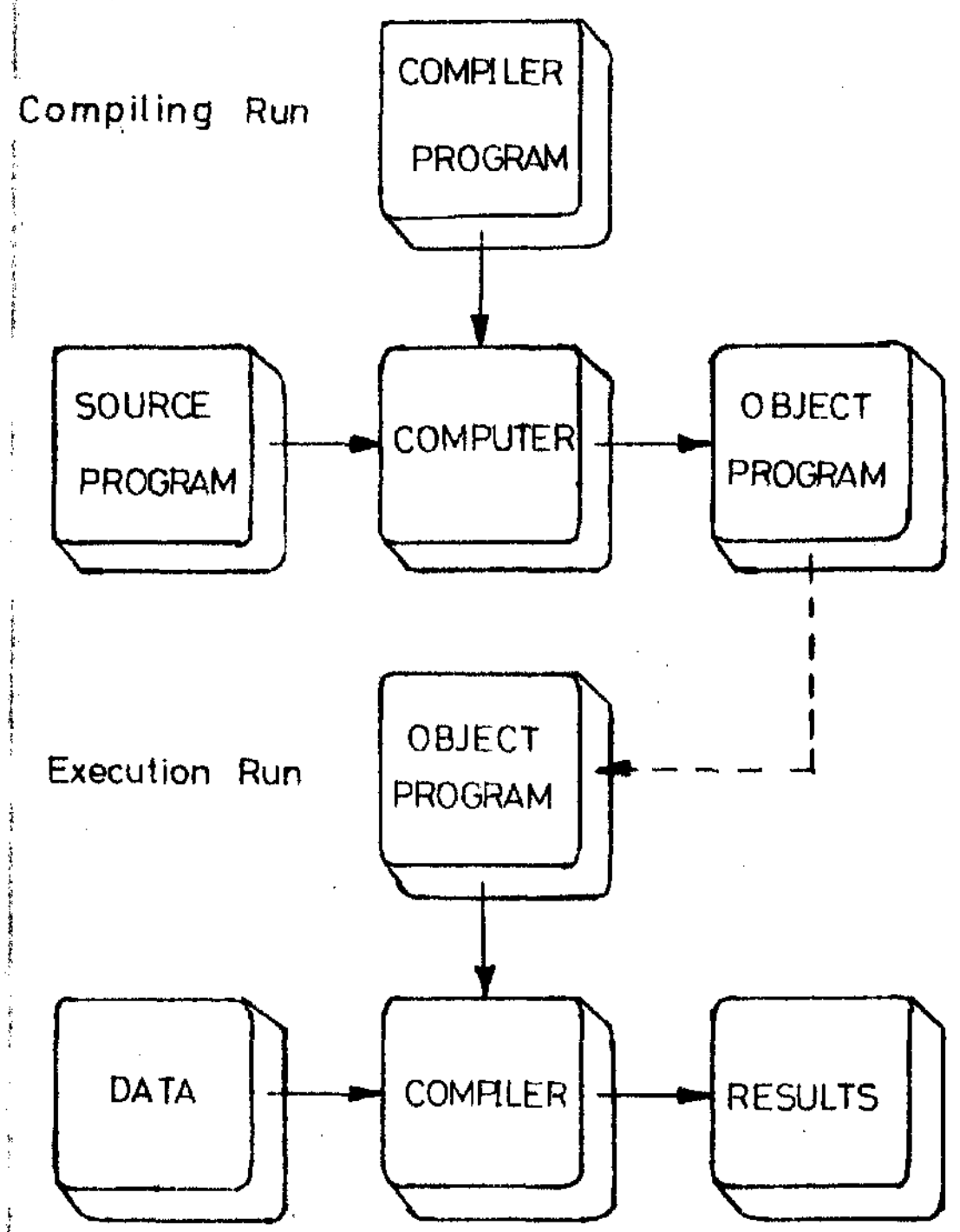


Figure 3,1 - Two-Step Execution Process

computer. Cross-compilers, however, are compilers which are hosted on one computer, but produce object code for another, usually different, computer. Thus while compilation of programs is carried out on the host computer, execution of object programs is carried out on the object computer. The transfer of the object code from the host machine to the target machine is achieved by automatic, semi-automatic or manual means such as magnetic tape, punched cards or keyboard entry respectively. Cross-compilers are implemented for computer systems (such as microcomputers) which cannot support resident compilers due to memory size limitations. In addition, cross-compilers support parallel development of system hardware and software.

The need for compilers and cross-compilers is obvious. With machine language, the programmer must communicate directly with a computer in terms of bits, registers, and very primitive machine operations. Since a machine language program is nothing more than a sequence of 0's and 1's, programming a complex algorithm in such a language is tedious and hence error prone. Perhaps

the most serious disadvantage of machine-language coding is that all operations and operands must be specified in a numeric code. Not only is a machine language program cryptic, but it may also be be impossible to modify in a convenient manner. A high-level programming language allows a programmer to express alrorithms in a more natural notation that avoids many of the details of how a specific computer functions.(22) For example, it is much more natural to write the expression A + B than a sequence of machine language instructions to add A and B.

A high-level programming language makes the programming task easier, but it also introduces some problems. The most obvious one is that a program is required to translate the high-level language into a language the machine can understand. Resident compiler programs are usually written in the machine or assembly language of the host computer. Cross-compiler programs have an advantage in that they can be written in a high-level language for which there is a compiler resident on the host computer.

The other major advantages of cross-compilers are that they may be tailored to reflect the target machine characteristics and limitations, and that they can be used for preliminary analysis of micro-computer systems before purchase or construction.

3.2. The Structure of a Cross-Compiler

A cross-compiler is similar in overall structure to a resident compiler, however, some differences exist due to the different applications. The discussion in the remainder of this chapter is restricted to cross-compilers for micro-computers.

A cross-compiler accepts as its input a source program and produces as output an equivalent sequence of machine instructions, either in the form of assembly code or in machine code. To achieve this, it must first perform an analysis of the source code, and from the verified source code, generate the equivalent object code. This process is complex, and is usually implemented logically in one or more separate stages.

It is customary to partition the compilation process into a series of phases (Figure 3.2). Each phase involves a logically cohesive operation that accepts as input a representation of the source program and produces as output a more advanced representation until finally a complete object code is produced as the output of the final phase.

The first phase, called the lexical analyser, separates characters of the source language into groups that logically belong together; these groups are called to tokens. The usual tokens are terminals, such as DO or parentheses, identifiers, such as X or NUM, and literals such as 0 and 105. The output of the lexical analyser is a string of uniform length symbols (tokens), which is passed to the next phase, the syntax analyser. The generation of the uniform length tokens is important, since it allows the compiler to achieve a uniform internal representation of the external source program, making it possible to change the lexis of the source language with minimal effect on the compiler. The lexical analyser also detects and records lexical

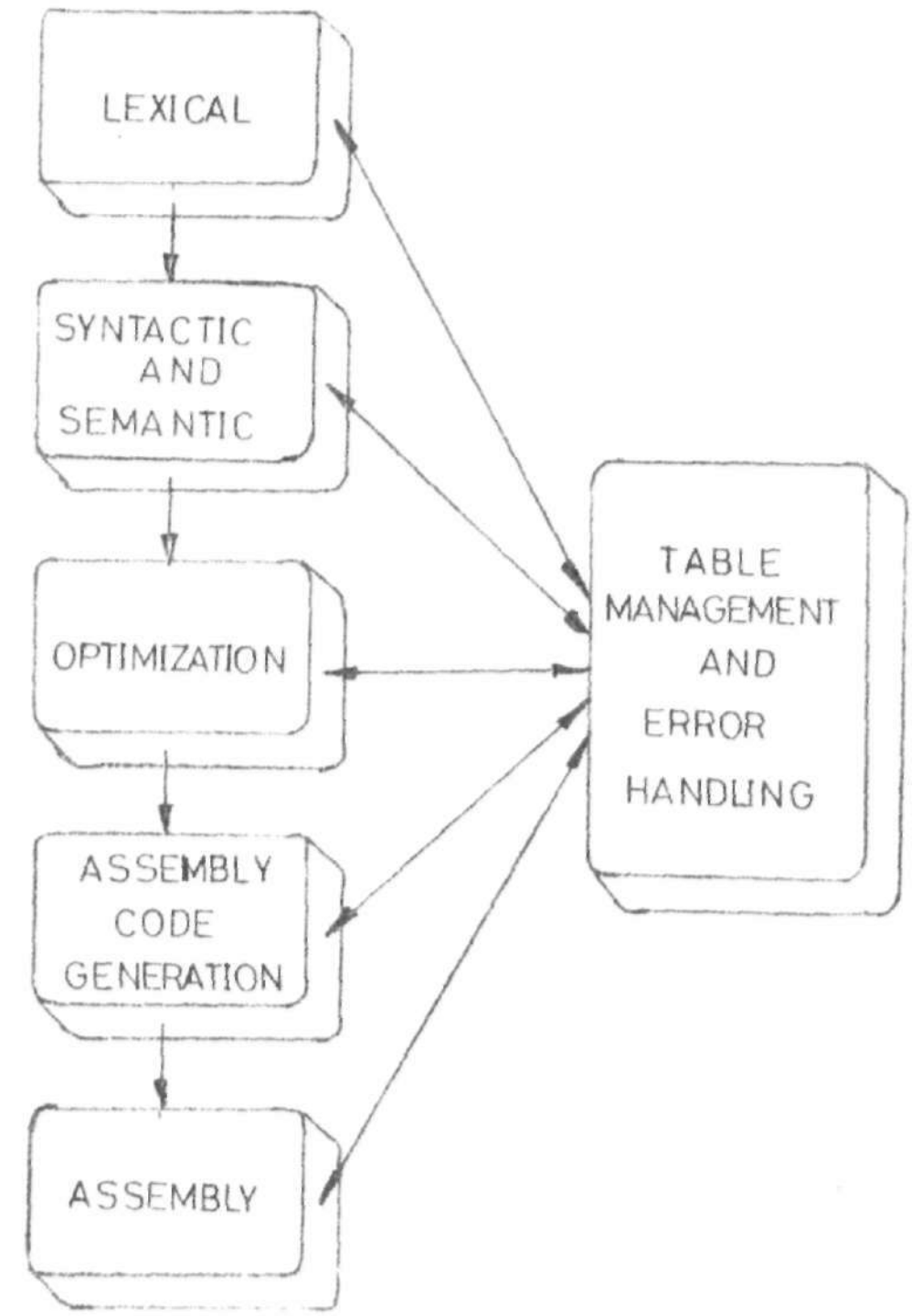


Figure 3,2 - Phases during Compilation

errors in the source program.

Phase two is the syntax analyser. This phase classifies tokens together into possible groups that can be identified with a particular syntactic structure. For example, the three tokens representing A + B might be grouped into a single syntactic structure called an expression. Expressions might further be combined to form statements. The general method of recognizing the basic syntactic constructs is by the use of rules (reductions),[21] which specify the syntax form of the source language. Upon recognising a construct, the syntax analyser calls the appropriate semantic (action) routine from intermediate code generator.

The semantic routines of the intermediate code generator use the structure produced by the syntax analyser to first interpret the meaning of the construct, and then generate a stream of simple instructions corresponding to the structure. Many styles of intermediate code are possible. One common style uses instructions

with one operator and a small number of operands, usually two. Optionally, some pointers may be implemented in the intermediate form which may be used for code optimization.[14,21,23,30,43] Code optimization is an important phase designed to improve the intermediate code so that the ultimate object program runs faster and/or takes less memory space. This is necessary in cross-compilers since memory limitations for microcomputers are severe, and due to their fabrication techniques, microcomputers are slow. Code optimization eliminates unwanted common subexpressions, memory accesses, etc. The output is another intermediate code program which is a more optimal representation of the source program.

The final phase, code generation, produces the object code by deciding on the memory locations for data, selecting code to access each datum, and selecting the registers in which each computation is to be done. A common way of implementing this stage is to first generate the assembly code from the intermediate code (machine dependent optimization) and then generate the

equivalent object code from the assembly code (i.e. assemble the assembly code).

3.2.2. Table Management and Error Handling

The table-management portion of the compiler keeps track of the names used by the program and records essential information about each, such as its type. The data structure used to record this information is called a symbol table.

The error handler is called when a block of the normalized source program does not fall within the class of acceptable constructs. It must warn the programmer by issuing a diagnostic, and adjust the information being passed from phase to phase so that each phase can proceed. It is desirable that compilation be completed on incorrect programs, at least through the syntax analysis phase, so that as many errors as possible can be detected in one compilation. Both the table management and error handling routines interact with all phases of the compiler.

The phases of a compiler may be grouped into two

categories namely machine independent phases and machine dependent phases.(21) The machine independent phases are the lexical, syntax, intermediate code generation and code optimization phases. These phases are said to be machine independent because they are not affected by the nature of the target machine. Their only dependence is on the source language, and as such, they are also called the language dependent phases. A change in the language would require changes in only the machine independent phases. The intermediate code serves as a link between the machine independent phases and the machine dependent phases.

The machine dependent phases are the code generation and assembly phases. The code to be generated clearly depends on the target machine, and not on the source language. These phases are also known as the language independent phases. A change of the target machine would require changes in only the machine dependent phases. In cross-compilers, many different target machines can be catered for by simply changing the the databases for the code generation stage. This

leads to a very powerful tool for reducing the development of useful codes for several machines. Clearly with a slight modification, it is possible to use the device for correspondence and hence for selection of the target μP. Other advantages include the ease of transfer, which could be achieved in a number of ways.

3.3. The Lexical Analyser

The lexical analyser is the interface between the source program and the compiler.(4) The lexical analyser reads the source program one character at a time, transforming the source program into a sequence of units called tokens. Each token represents a sequence of characters that can be treated as a single logical entity. Identifiers , keywords, constants, operators, and punctuation symbols such as commas and parentheses are typical tokens. For example, in the FORTRAN statement

```
IF (M.EQ.5) GOTO 100
```

the following eight tokens are found: IF; (; M; .EQ.;

5;); GOTO; 100. It is seen that a token is a sub-string of the source program that is to be treated as a single unit.

There are three kinds of tokens: spe•ific strings (reserved words or part of the language) such as IF or a semi-colon (called terminal symbols), numbers such as constants (called literals) and variable names such as M (called identifiers). A token is usually treated as a pair consisting of two parts: a token type and a token value. Thus in the example above, M has type identifier and value 'M'. A usual way of realizing the lexical phase is by use of tables called symbol tables. Each entry in these tables has a value, and optionally, attributes.

The terminal table is a permanent data-base that has an entry for each terminal symbol (e.g. arithmetic operators, keywords, non-alphanumeric symbols). Each entry consists of the terminal symbol. This table is defined by the source language and is used by the lexical phase to recognize terminal symbols. Terminal

symbols which are not keywords are called break characters.

The literal table is created by lexical analysis to describe all literals used in the source program. There is one entry for each literal, consisting of a value, and optionally a number of attributes, such as whether the literal is a constant, the literals' data type, etc.

The identifier table is also created by lexical analysis to describe all identifiers used in the source program. There is one entry for each identifier. Lexical analysis creates the entry and places the name of the identifier into that entry. Optional attributes may include type information, etc.

The label table is created by the lexical phase to describe all labels used in the source program. There is one entry for each label. Lexical analysis creates the entry and places the name of the label into that entry. Where labels are numeric, the label table

may be implemented as part of the literal table.

The uniform symbol table is created by lexical analysis to represent the program as a string of tokens rather than of individual characters.(21) Spaces and comments in the source code are not represented by uniform symbols and are not used by future phases. There is one uniform symbol for every token in the program. Each uniform symbol table entry contains the identification of the table of which that token is a member, and its index within that table. Thus the uniform symbol table is a series of pointers to the other tables which contain the values of the tokens.

There are many ways of implementing the lexical phase. In one method, the input string is separated into tokens by break characters. Source characters are read in, checked for legality, and tested to see if they are break characters. Consecutive nonbreak characters are accumulated into tokens. Strings between break characters are tokens, as are nonblank break characters. Blanks may serve as break characters but are otherwise ignored.

Lexical analysis recognizes three types of tokens: terminal symbols, possible identifiers, and literals. It checks all tokens by first comparing them with the entries in the terminal table. Once a match is found, the token is classified as a terminal symbol and the lexical analyser creates a uniform symbol table entry of type, say 'TRM' and index equal to the index of that terminal symbol within the terminal symbol table. If a token is not a terminal symbol, lexical analysis proceeds to classify it as a possible identifier or literal. Those tokens that satisfy the lexical rules for identifiers are classified as 'possible new identifiers', and the identifier table is examined. If this particular token is not in the table, a new entry is created. Regardless of whether or not an entry had to be created, lexical analysis creates a uniform symbol entry of type say 'IDN' and inserts it into the uniform symbol table, together with an index.

Numbers are classified as literals. After a token has been classified as such, the literal table is

examined. If the literal is not yet there, a new entry is made. Regardless of whether or not a new entry is made, a uniform symbol entry of type say 'LIT' is created and put into the uniform symbol table.

Some programming languages require the explicit declaration of all variables, their types, labels and constants. The compilers for these types of languages have simpler lexical phases, since all the table construction may be completed by the end of the declarative part. What remains is the construction of the uniform symbol table.

The lexical analysis may make one complete pass over the source code and produce the entire uniform symbol table.(21) Another scheme is to have the lexical analyser called only when the syntax phase needs the next token.(29) In either case, the uniform symbol table is the input to the syntax analyser.

3.4. The Syntax Analyser

The function of the syntax analyser is to recognize the major constructs of the language and to call the appropriate semantic routines that will generate the intermediate form for these constructs.(4)

Major constructs of the language are recognized by interpreting general rules called reductions.(21) These reductions are dependent on the syntax of the source language, and must be modified if the syntax of the source language is changed. Although it is more efficient to code each reduction in the implementation language than to have an interpreter for the reductions, writing the reductions is helpful in visualizing the problem. Reductions are therefore considered to be one of the databases of the syntax analyser. The other databases in this phase are the stack and the uniform symbol table.

The stack is the collection of uniform symbols that is currently being worked on by the syntax analysis and

intermediate code generation phases. Additions to or deletions from the stack are made by the phases that use it. Each symbol from the uniform symbol table enters the stack only once.

The syntax analyser compares the contents of the stack with each reduction in turn. When a match is found, the appropriate semantic routine which handles that construct is called from the intermediate code generator.

The purpose of the semantic routines is to create an intermediate form of the source program. The separation of the syntactic phase from the semantic phase is a logical division. The former phase recognizes syntactic constructs while the latter interprets the precise meaning into the matrix. In practice, it may not be obvious where the recognition of a construct ends and the interpretation of its meaning begins. This may be illustrated by the problems of handling arithmetic expressions. An entire arithmetic expression may be considered a syntactic entity, so that its

recognition should be a problem for the syntax analyser, and its interpretation a problem for a semantic routine. However, each subexpression (two operands and an operator) can also be considered a syntactic entity and therefore should be recognized by the syntax analysis phase. For the first case, the semantic routine may construct matrix entries for the entire expression, and for the second case the semantic routines would only construct one entry for for a subexpression. The tradeoff is between how much computation is performed by the syntax phase versus how much by the action routines.

The primary intermediate form of the program is the matrix.[21,23,30] There are two reasons for having a matrix: (1) it allows the first four phases of the compiler to be machine-independent; (2) it allows machine-independent optimization of the program to occur before the code generation phase. A simple form of a matrix entry consists of a triplet where the first element is a uniform symbol denoting the terminal symbol or operator and the other two elements

are uniform symbols denoting the arguments.(21) The code generation phase will be driven by this matrix and will produce the appropriate object code.

There may also be fields reserved for retaining of entries that can be utilized by the optimization phase to add or delete entries. There are four commonly used types of machine-independent optimization techniques, These are: (1) Elimination of common subexpressions - in which duplicate matrix entries are eliminated; (2) Compile time compute in which computations involving constants are executed at compile time; (3) Boolean expression optimization - in which properties of Boolean expressions are used to shorten their computation,and (4) Movement of invariant computations outside of loops - in which computations that remain invariant in a loop are removed from the loop.

In deciding whether or not to incorporate any of these optimization schemes into a compiler, it is necessary to weigh the gains it would bring in increased

efficiency for the object program against the increased cost in compilation time and complexity. The outcome of the decision should depend heavily on the circumstances under which the compiler is being produced and on the use to which it will be put. For microcomputer cross-compilers, there is a definite gain since code execution is faster, and space reduction increases the optimization further.

The machine independent phases of the compiler having been executed, the optimized form of the matrix is passed on to the machine dependent phases.

3.5. The Machine Dependent Phases

The machine dependent phases of the cross-compiler are the most important aspects for such translators. In particular, these phases must reflect the characteristics of the target microprocessor, such as the architecture, special data structures such as stacks, and the speed and storage constraints.

The machine dependent phases comprise the code generation and assembly phases, and include the machine dependent optimization and storage assignment stages.

The purpose of the code generation phase is to produce the appropriate code (assembly or machine language). The code generation phase has the matrix as input, and uses code productions to generate code. It also references the identifier and literal tables in order to generate proper address and code conversion.

Code productions are a permanent data base defining all possible matrix operators. The code generation process is implemented in a way analogous to that used for an assembler macroprocessor. The operation field of each matrix line is treated as a macro-call, and the matrix operands on the line are used as macro arguments. Each matrix line then generates code according to the macro definition.

Code generated in this manner is usually not optimal with respect to register usage. Many unnecessary load

and store operations are performed, and this is not desirable on a microcomputer. To avoid these redundant loads and stores, a code generator might keep track of the run-time contents of registers. Knowing what quantities reside in registers, the code generator can generate loads and stores only when necessary. This is known as machine dependent optimization, and is particularly important for microprocessors where memory limitations are usually severe and high-speed registers are few. A wise choice of machine instructions is also desirable.

The storage assignment stage assigns storage to all variables and literals referenced in the source program. It also assigns storage to all temporary locations that are necessary for intermediate results, e.g. during evaluation of arithmetic expressions. These storage references were reserved by the intermediate code generation phase and do not appear in the source code. The availability of registers and the modes of memory addressing are important factors in the design of this stage.

The last phase in a cross-compiler is the assembly phase. The task of this phase depends on how much has been done in code generation. If a lot of work has been done in code generation, then the assembly phase must resolve label references in the object program and format the object deck. If on the other hand the code generation has simply generated symbolic machine instructions and labels, the assembly phase must resolve label references, calculate addresses, generate binary machine instructions and generate storage and convert literals.

In cross-compilers, the advantage in separating the machine independent and the machine dependent phases lies in the ease of catering for different micro-processors. Since there are little limitations on compilation time, the compiler may be slow, and may attempt to provide good error diagnostics and perform some code optimization.

4. IMPLEMENTATION

'Would you tell me, please, which way I ought to go from here?'
'That depends a great deal on where you want to get to,' said the Cat.

LEWIS CARROLL
Alice's Adventures in Wonderland

4.1. Introduction

An overall picture of the implementation of the cross-compiler for MULANG I is presented in this section. Later sections discuss the various phases of the cross-compiler.

The cross-compiler was written in Minnesota FORTRAN (MNF) Version 5.0. MNF and its associated compiler were developed by the staff of the University of Minnesota Computer Center. It is designed for use in batch and time-sharing applications on the Control Data Corporation (CDC) 6000/7000/Cyber Series of computers, running under the SCOPE, KRONOS or NOS operating systems. The American National Standards Institute (ANSI) FORTRAN X 3.9-1966 is a subset of MNF.

'Minnesota FORTRAN is a fast-compiling FORTRAN featuring good execution code, extensive compilation diagnostics, complete cross-reference maps, and many convenient execution-time debugging aids.'(25)

In addition, MNF features a good character handling capability, and structured constructs like the IF-THEN-ELSE-ENDIF. Ease of use and familiarity were deciding factors in the choice of MNF as the implementation language.

The host machine is the mainframe at the Ahmadu Bello University Computer Centre. This is a CDC Cyber 72-13 Central Processor with 48K, 60 bit words of core storage. The peripherals feature 4 x 110 Mbit disk storage units, 2 magnetic tape drives, 24 interactive terminals, 2 card readers, and 2 line printers. The present configuration is shown in Figure 4.1. The machine presently runs under the Network Operating System (N.O.S.) Version 1.1.0. This is a vendor supplied operating system which supports both time-sharing and batch processing.

The target machine is a M6800 prototyping system in the Ahmadu Bello University Electrical Engineering Department. This is based on a Motorola 6800 microprocessor and its present configuration is discussed

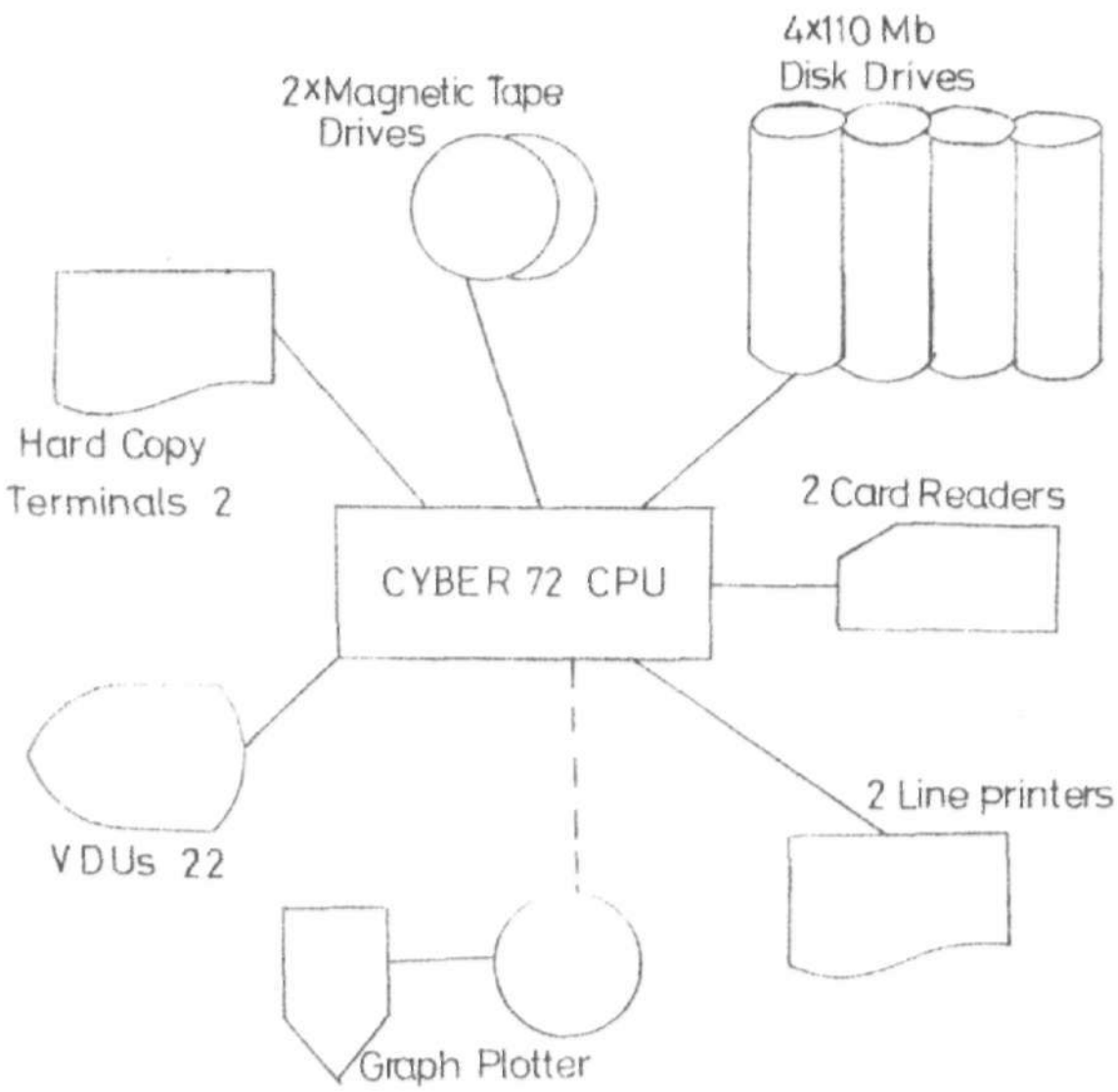


Figure 4.1-Mainframe Hardware Configuration

in Appendix A of this thesis.

4.1.1. Cross-Compiler Structure

The MULANG I cross-compiler is a software package which is implemented in two stages. The first stage is a semicompiler stage which accepts as input source programs written in MULANG, and produces as output equivalent M6800 assembly code. The second stage is a code generator which accepts as input formatted M6800 assembly code, and produces as output machine code for the M6800. This is shown in Figure 4.2.

The reason for this design was to allow the cross-compiler package to be used as an assembler if so required. Thus programs meant for the M6800 could be written in MULANG, or in M6800 assembly code. One advantage of this scheme is that it allows the programmer to recode critical sections of his program in a more optimal manner. Assembly code produced by the semicompiler stage could be modified using the text editing facilities available under NOS. The resultant code could then be assembled to produce improved machine code for transfer to the M6800.

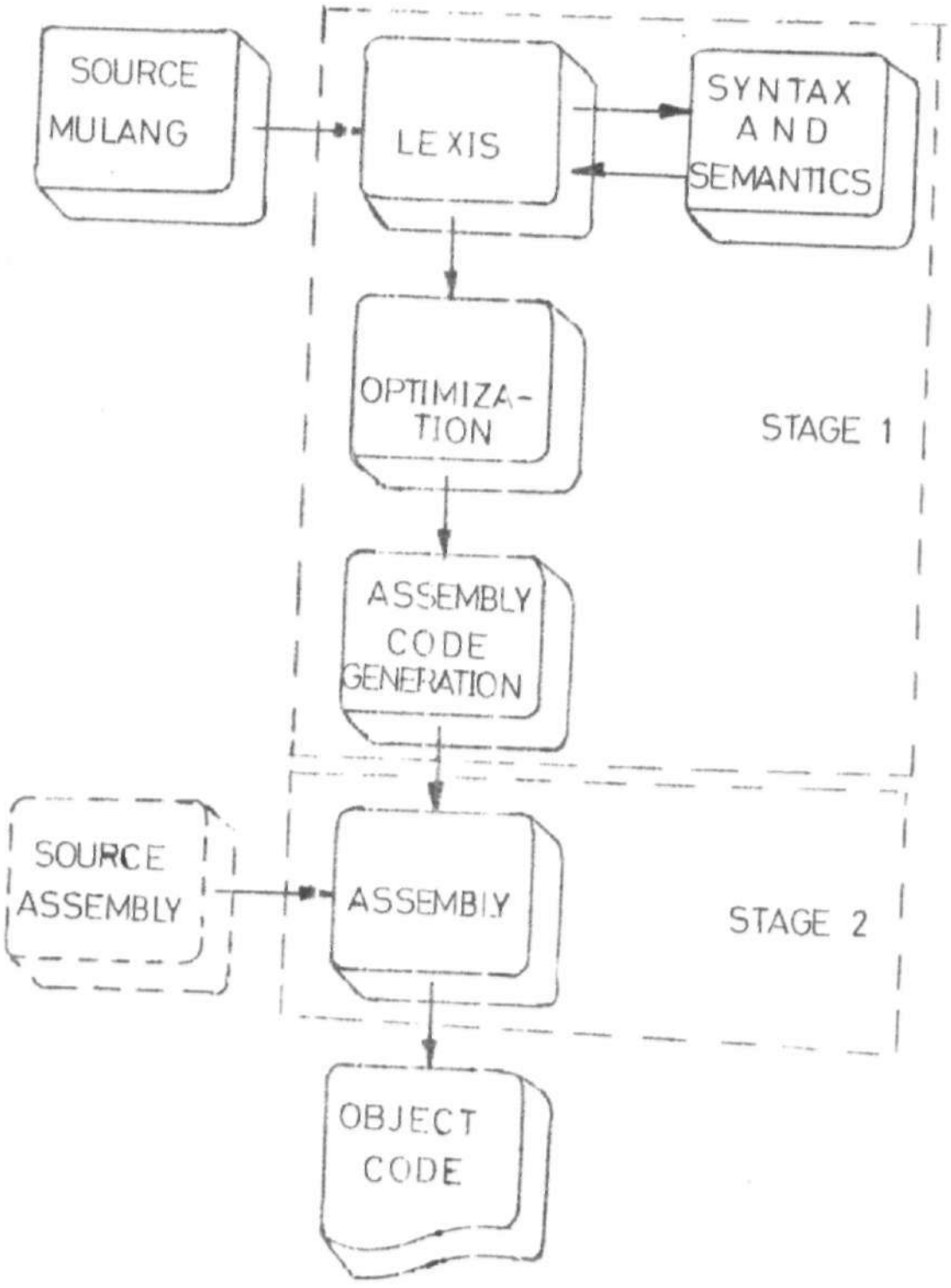


Figure 4,2 - Cross-Compiler Structure

In general, assembly language can be useful when programming machine-dependent features (such as interrupt-driven input/output devices), functions in which time is at a premium, and critical loops - loops which require small amounts of memory, but a disproportionately large amount of execution time.[18] Perhaps the most common instance in which assembly language is needed is when a programmer wants to do something the high-level language was not designed to do. For example, assembly code would be required to implement bit handling, which cannot presently be done using MULANG I.

The semicompiler stage incorporates four compilation phases: lexical; syntactic and semantic; optimization; and assembly code generation. The assembly stage incorporates one phase - that of machine code generation or assembly. Each phase is implemented as a suite of routines, called by a master routine whenever they are required. The passes of the semicompiler are four in number. The first pass incorporates the lexical/syntactic and semantic phases. The input to this pass is

the source program in MULANG I, and the output is an intermediate form of the source program. The second pass accepts as input this intermediate form and performs some optimization. The output of this pass is a modified version of the intermediate form. The third pass accepts as input the optimized intermediate form and generates equivalent assembly code. The fourth and final pass accepts the assembly code program and produces an equivalent M6800 machine code program.

4.1.5. Cross-Compiler Usage

Source programs in MULANG I or M6800 assembly language must be created as files, which are input to the cross-compiler or assembler respectively. The operating system on the host machine supports comprehensive facilities for file creation in a time-sharing mode. Alternatively, jobs may be run using card decks.

After compilation, the source program listing, error messages, if any, and the machine code output are printed onthe line printer. The machine code must be entered manually into the M6800 prototyping system, due to the absence of a suitable interface.

It is envisaged that a comprehensive user reference manual shall be written in the near future.

4.2. The Machine Independent Phases

The machine independent phases of the cross-compiler are the lexical, syntactic and semantic, and optimization phases. These phases have been implemented as suites of subroutines. The lexical scanner reads the source program and creates databases, while the syntax and semantic analysers generate the intermediate form of the source program.

Each line of the source program is input sequentially to the cross-compiler. The lexical scanner, controlled by a master routine, reads that line as a string of characters, and forms tokens by searching for break characters. If the first column of a line contains '?', then the lexical scanner ignores that line and proceeds to the next line. Starting from column 1, the scanner searches for the next break character. When that break character is located, the scanner packs

together all characters between the present break character and the last, and forms a word. A check is made to see that the word conforms to the specifications for a symbol of MULANG I. If the word is illegal an error message is output. If the break character is the first break character on the line, a check is made to see if that break character is a colon. If it is, the first word is classified as a label, and a check is made against the label table. If the check is unsuccessful an error message is generated, but an attempt at recovery is made by inserting the word onto the label table and making an appropriate entry into the uniform symbol table.

The requirement by MULANG I for the mandatory declaration of all variables, constants and labels simplifies considerably the scanning process. The identifier, constant and label tables are set up by the end of the declarative part, and all future non-recognizable symbols cause the generation of error messages.

During processing, break characters are located and

strings between consecutive break characters are packed to form words. Each word is compared first to entries in the terminal table - which contains keywords and break symbols. If a match is found, an entry is made into the uniform symbol table. If there is no match, a check is made to see if the word is a constant or an identifier. Depending on the result of the check, the constant or identifier tables are searched respectively. If a match is found, an entry is made into the uniform symbol table, else an entry is made both into the constant or identifier table, and the uniform symbol table, and an appropriate error message is generated.

This attempt at error recovery allows the cross-compiler to finish processing the entire program. Recovery insertions may cause syntactic and semantic errors, as a consequence of incorrect classification of the word. These errors will, however, be detected by later phases and appropriate error messages generated.[11,16]

When the lexical scanner recognizes that an entire statement has been read (note that a statement may occur over one or more lines, and is delimited by a ';') it calls the syntax and semantics phase. The input to this phase is a uniform version of the source program statement scanned by the lexical phase. This method of calling the syntax and semantic phase upon recognition of a statement by the lexical phase was implemented to conserve storage space. An alternative scheme would have been to pass the entire source program through the lexical analyser before calling the syntax and semantic analyser.

The syntactic and semantic phase is implemented in a manner similar to the lexical phase in that a master routine (which is called by the scanner) controls execution through this phase. It receives the inputs from the lexical phase, and attempts to recognize each statement presented to it. Upon recognition of a statement, it passes control to a smaller dedicated syntactic routine which checks the syntactic correctness of the statement. If the statement is found to

be correct in syntax, an appropriate semantic routine is called. The semantic routine accepts the input statement, and processes it. This may require action by one or more semantic routines, and causes insertions to be made in the intermediate form of the matrix.

The concept of reductions, mentioned in Chapter 3, was used in the design of the syntactic routines, while the action routines in the reductions were implemented as the semantic routines. This method follows closely one presented by Donovan.(44) The general form of the reductions is:

Label: Old Top of Stack/Action Routine/New Top of Stack/Next

The following conventions are used

1. Label - optional
2. Old Top of Stack - to be compared to actual Top of Stack.
   (a) Blank or null - always a match, regardless of what is on the stack
   (b) Non blank - one or more items from the

following categories:

(1) < syntactic type > such as identifier or literal - matches any uniform symbol of this type

(2) < any > - matches a uniform symbol of any type

(3) symbolic representation of a keyword, such as 'IF' or ';' - matches only the uniform symbol for this keyword.

3. Action Routines - to be called if Old Top of Stack matches Top of Stack:

(a) Blank or null - no action routines called

(b) Name of action routine - call the routine.

4. New Top of Stack - changes to be made to Top of Stack after action routines are executed:

(a) Blank or null - no change

(b) " - delete Top of Stack (i.e. pattern that has been matched)

(c) Syntactic type, keyword or stack item (Sm) - delete Old Top of Stack (i.e. pattern that has been matched in 2) an replaced with this item(s)

(d) * - get next uniform symbol from uniform symbol table and put on top of stack

5. Next - next reduction

(a) Blank or null - interpret the next sequential reduction

(b) n - interpret the reduction with label n.

Within the reduction stack fields with the oldest item are written on the left and the newest on the right. $S_m\ S_{m-1}.....S_2\ S_1$ denotes a field m long, where S_1 refers to the top of the stack, S_2 to the second item on the stack, etc.

During the syntax and semantic analysis phase, items from the uniform symbol table are placed into a stack. The stack, starting from the top (actual Top of Stack) is compared with the Old Top of Stack of each reduction in turn, until a match is found. When a match is found, the action routines specified in the action field are executed in order from left to right. When control returns to the syntax analyser, it modifies the Top of Stack to agree with the New Top of Stack field.

Step 1 is then repeated with the reduction specified in the next reduction field. The reductions used in the implementation are shown in Figure 4.3.

This method of implementation allows easy detection of syntax errors, before any intermediate code is generated by the action, or semantic, routines.

The primary intermediate form of the source program is the matrix, which is implemented as a 5 element wide array. There are two reasons for having a matrix: (1) it allows the required separation between machine independent and machine dependent phases; (2) it allows machine-independent optimization of the program to occur before the code generation phase.

A simple form of a matrix entry consists of a triplet, where the first element is a uniform symbol denoting the terminal symbol or operator, and the other two elements are uniform symbols denoting the arguments or operands.

```
 1                          /          / **
 2   PROGRAM   idn          /progbgn   / S1*   /5
 3   SUBPROGRAM idn         /subbgn    / S1*   /5
 4   any    any             /error     / S1*   /2
 5   INTEGER                /intproc   /**     /
 6   LOGICAL                /logproc   /**     /
 7   LABELS                 /labproc   /**     /
 8   CHANNELS               /chanproc  /**     /
 9   CONSTANTS              /conproc   /**     /11
10     any                  /error     /***    /5
11     idn  :               /labelproc/***    /
12     idn  =               /aritnop   /***    /
13   CALL   idn             /callproc  /***    /
14   JUMPTO   idn           /jumpproc  /***    /
15   IF    (                /ifproc    /***    /
16   ELSE   ;               /elseproc  /***    /
17   ENDIF  ;               /endifproc/***    /
18   DO    idn              /doproc    /***    /
19   ENDDO  ;               /enddoproc/***    /
20   WHILE  (               /whileproc/***    /
21   ENDWHILE  ;            /endwproc  /***    /
22   INPUT   (              /inputproc/***    /
23   OUTPUT  (              /outputproc/***   /
24   END   idn              /endprog   /***    /
25   END PROGRAM            /stop      /       /
26     any  any             /          /       /11
27     any  any             /          /       /3
```

Figure 4.3. Reductions for MULANG

However, the matrix was implemented as a 5 element wide array to allow optimization of the intermediate form of the source program. The first 3 entries of the matrix comprise the triplet mentioned above, while the other 2 entries are forward and backward pointers respectively, to aid optimization.

The forms of optimization attempted in this implementation were the elimination of common subexpressions, and compile time compute. The matrix is scanned sequentially from top to bottom, and equivalent subexpressions recognized. One (the second) of the subexpressions is then ignored by adjusting the pointers of both entries. Similarly, constant computations are also recognized in the same pass, and they are computed. Matrix entries are adjusted to reflect this computation.

4.3. Translation of Arithmetic Expressions

Arithmetic expressions are an important part of most computer languages, so their translation is a part of

the compiler's job. The technique employed for the translation of arithmetic expression in the MULANG I cross-compiler is described below.

Almost all algebraic operations are binary, so a useful concept is the algebraic triplet of elements, consisting of the two operands and the result

$$E_C = E_L \ominus E_R$$

where

E_C : result

E_L : left-hand operand

$\ominus$: operator

E_R : right-hand operand.

Unary operations, such as negation, can be considered to be special cases of binary operations, where E_L is assumed to have a special value and plays no role in the calculation. For example, if the negation operator is used, $\ominus$ would be subtraction and $E_L = 0$. The translator's task, given an expression, is to find a sequence of triplet operations that, when executed with the values of the variables defined, calculates

calculates the value of that expression. Results E_c of intermediate triplets may be held as intermediate variables T_i (i = 1 .....), which are not declared or used in the program in the program itself, but defined and used by the translator during its processing. For example,

$$a = b + c \times d$$

has as its set of triplets

$$T_1 = c \times d$$
$$a = b + T_1$$

since the value of c x d must be found before addition to c.

Intermediate variables T_i need not be stored in memory - they may be held in registers, ready for use in future calculations. The limitation is the number of available registers, which are expensive and therefore few. Since the number of intermediate variables is unknown in advance, a sophisticated translator will optimize the use of intermediate (temporary) storage during execution.

For example, the expression

$$a = \big((b + c) \times (d - e)\big)/(f - g)$$

could be evaluated by the following string of triplets

$$T_1 = b + c$$
$$T_2 = d - e$$
$$T_3 = T_1 \times T_2$$
$$T_4 = f - g$$
$$a = T_3/T_4$$

However, a more optimal use of temporary storage would be the use of the following sequence of triplets

$$T_1 = b + c$$
$$T_2 = d - e$$
$$T_1 = T_1 \times T_2$$
$$T_2 = f - g$$
$$a = T_1/T_2$$

In a machine with a limited number of registers, some intermediate variables may have to be stored in memory. This is not desirable in terms of both memory usage and execution time. For example, in a two register machine such as the M6800, the latter

calculation might be much faster than the former, saving 4 memory accesses out of 11. If this expression were in a loop, cycling thousands of times, a saving in time of nearly 40 percent would be significant. The need for optimization of temporary storage is obvious, and much work has been carried out in this field.[1,3,11,13,36,54,65]

One of the earliest algorithms for the translation of arithmetic expressions, was one due to Rutishauser (1951).[62] In his algorithm, all expressions were assumed to be fully parenthesized, and no implicit precedence of operators was assumed. Thus the expression

a = b + c x d

was illegal. The precendence rule requiring that the multiplication be performed first in the above expression was to be shown explicitly by parentheses:

a = b + (c x d)

The requirement for explicit statement of operator precedence in each expression by the use of parantheses

was cumbersome, and led to very long expressions. It is for this reason that the method of Rutishauser was not used in the implemention of the MULANG I cross-compiler.

The technique used is one due to Bauer and Samuelson (1961).(in 49) This technique makes use of two push-down stacks, and a transition table. The translator stack T is used directly as it translates the expression and the execution stack E is used during excution. The translator reads the expression once, from left to right, and outputs a sequence of instructions of two forms:

K_I - take the operand whose name is I and put it on the stack E

$K_@$ - take the top two operands from the stack E, operate on them with @, and store the result on top of stack E.

Symbols are then read in sequentially from the input string. If the symbol read is an operand I, the instruction K_I is output, and the translator moves on to the

next symbol. If the symbol read in is an operator @ one of several action is taken, the particular one being determined by the transition table. This table has rows and columns corresponding to operators in the language. The entries of the table contain the directions to the translator. There are four possible actions to be taken by the translator after reading an operator @. (The operator on top of T is designated by &)

1. Push @ on T; Read next symbol
2. Generate $K_{\&}$; Replace & by @; Read next symbol
3. Pop T; Read next symbol (used to delete parentheses)
4. Generate $K_{\&}$; Pop T; Repeat with same input symbol.

There are two other special instructions, End of Process and Error, which instruct the translator to stop and output the proper instruction.

The processor reads the table using as one index &, the top operator in T, and, as the second, @, the

operator last read off the input string. Figure 4.4 is the table used for the simple arithmetic operator precedence permitted in MULANG I. In the table, ∅ stands for the blank symbol, or empty string. If the stack T is empty, ∅ is considered to have been read. Similarly, if the input string is empty, ∅ will be considered read.

As an example, the expression

(a x b + c x d)/(a - d) + b x c

would cause the following sequence of instructions to be generated:

$K_a, K_b, K_x, K_c, K_d, K_x, K_+, K_a, K_d, K_{-1}, K_/, K_b, K_c, K_x, K_+$

However, a simpler representation results from dropping the K's

ab x cd x + ad - /bc x +.

This is a parenthesis free form called the Polish form. The executing program uses a push-down stack E and reads the Polish form from left to right. If it reads an operand, the operand is pushed on E; if an

Top of Stack ↓ / Input Symbol ●	$	(	+	-	*	/	**	)
$	End	1	1	1	1	1	1	Error
(	Error	1	1	1	1	1	1	3
+	4	1	2	2	1	1	1	4
-	4	1	2	2	1	1	1	4
*	4	1	4	4	2	2	1	4
/	4	1	4	4	2	2	1	4
**	4	1	4	4	4	4	1	4

Figure 4.4. - Transition Matrix for Operations Precedence

operator, the operator is applied to the top two elements of E, and the result is stored as the new top of E, after deleting the two operands. In the form produced by the translator, there are no implicit or explicit problems of precedence, since the program is executed directly from left to right. Temporary storage is automatically controlled by use of the push-down stack E, which holds all temporary intermediate values.

Matrix entries for arithmetic expressions are made from the execution stack E.

4.4. Machine Dependent Phases

The machine dependent phases of a cross-compiler are the code generation (assembly) and output phases. These phases have been implemented in two stages: the first phase, that of code generation from the intermediate form of the program, is a part of the semi-compiler stage mentioned earlier; the second and third phases, those of assembly and output, are implemented

as a separate program. This has the effect of allowing users to program in M6800 assembly code if they so wish.

The inputs to the code generation phase are the matrix, the identifier table, the label table, the constant table, and the terminal table. The data-bases used in this phase are the M6800 assembly code instruction set, and a set of macro like routines which substitute assembly code for matrix entries.

The matrix is scanned sequentially from top to bottom, and different types of matrix entries are recognized. The appropriate generation routine then substitutes the equivalent assembly code. One matrix entry may correspond to one or more assembly language lines of code. As an example, consider the matrix entries

```
+   A   B
=   C   b          b : blank
```

The entries are read from left to right. The plus sign in the first line causes the generator to generate an instruction to load A into the accumulator, and then

to add B to the contents of that accumulator. The second line would require the contents of the accumulator to be stored in C. The resultant code may look like

```
LDA   MA
ADD   MB
STA   MC
```

where MA, MB and MC are memory locations containing the values of A, B and C respectively.

In this manner, the program is transformed from the intermediate form into the assembly language form. The output from this stage is written onto a file, in the format shown in Figure 4.5. This format was chosen so as to allow ease of use of the assembler only.

The assembly phase accepts as input a file containing the assembly language program. This file may be the assembly output from the semicompiler with no changes, or a user written program in assembly language. The only database used by this phase is a table of assembly



Figure 4.5-Assembler Input Format

language instructions and their machine language equivalents.

This phase scans through the statements of the program, and substitutes machine language for assembly language, by comparing the assembly language instruction with the transformation table. Memory allocation starts from zero-address, unless otherwise specified by the user.

Machine code generated by the assembly phase is collected and printed in a readable form. This form can be transfered manually to the M6800.

It should be noted that since no analysis is performed in the assembly phase, user modified or user written assembly language programs which are entered to the assembler must be lexically, syntactically and semantically correct.

A change in microprocessor may be catered for by first changing the assembly code generation routines, and then the assembly language to machine language database.

5. CONCLUSIONS

We ain't what we want to be, and we ain't what we're goin' to be, but we ain't what we wuz.

SOUTH CAROLINA Mountain Proverb

5.1. Conclusion

The prototype of MULANG developed in this thesis - MULANG I - is a working model, and may be used for practical applications. It compares favourably(in terms of ease of use) with languages developed elsewhere - by both educational[13,19] and commercial[51,68] institutions. Most programming languages available for microcomputers are PL/1 derivates, such as PL/M (Intel), PLZ (Zilog), PL/65 (Rockwell) and MPL (Motorola). These have resident compilers which require much memory and hence more expensive development systems. Newer languages such as FORTH and UCSD Pascal are now available[51] but these do not use compilers in the strict sense of the word. They are interpreted.

To make MULANG compatible with the commercially available languages, some improvements are necessary, the major of which would involve the provision of more data types. At present, only logical and integer data types are catered for. It is felt that MULANG should

provide for at least real and character data types as well. Real data types would permit applications requiring real numbers, and floating point arithmetic (e.g. in measuring instruments). Character data types can be used in applications which would require extensive character handling (e.g. in 'smart' video terminals). One other data type which is very essential is the bit data type. A need is foreseen of some applications for extensive bit handling (e.g. I/O interfacing), and the present version of MULANG does not provide for bit manipulation.

Having provided new data types, the logical extension would be to provide data structures such as stacks, arrays, lists, pointer structures, and complex numbers. Future versions of MULANG could also permit recursion, to improve programming and reduce memory usage. Addition of these features would allow the microcomputer to perform tasks traditionally reserved for minicomputers.

There is also a need to improve the cross-compiler. The first step would be to rewrite the entire cross-compiler in a language suitable for writing compilers. Pascal and Simula are both good implementation languages for compilers, and both are available at the Ahmadu Bello University Computer Centre. Other improvements in the cross-compiler would arise from implementing further error detection (semantic) and recovery, and from providing substantial optimization of memory usage and execution times.

The cross-compiler system as a whole could be improved by providing a link between the mainframe and an ASR.33 teletype avaialable in the Electrical Engineering Department. The teletype is equipped with a paper tape punch, and this could serve as the interface, to reduce the possibility of errors during human transmission of object code from the host to target machines. Moves are already being made in this direction.

The cross-compiler developed herein forms the foundation of a proposed microprocessor cross-development

system. Additions envisaged include a cross-assembler with macroassembly features, a simulator to enhance debugging, and a linker/loader to permit linking and loading of programs written in both MULANG and assembly language. (Figure 5.1).

An attempt has been made in this thesis to design a language, and a cross-compiler for that language. It is believed that this is the first such attempt in any Nigerian University. There is a need to develop products indigenously, and it is hoped that the advent of MULANG will make the development of products employing microprocessors easier.

The Intergovernmental Bureau for Informatics,[72] and other bodies,[69,70,71] have established that whereas hardware product development in developing nations may be circumscribed by high capital costs, software product development could, and should be on par with that in industrially developed nations.

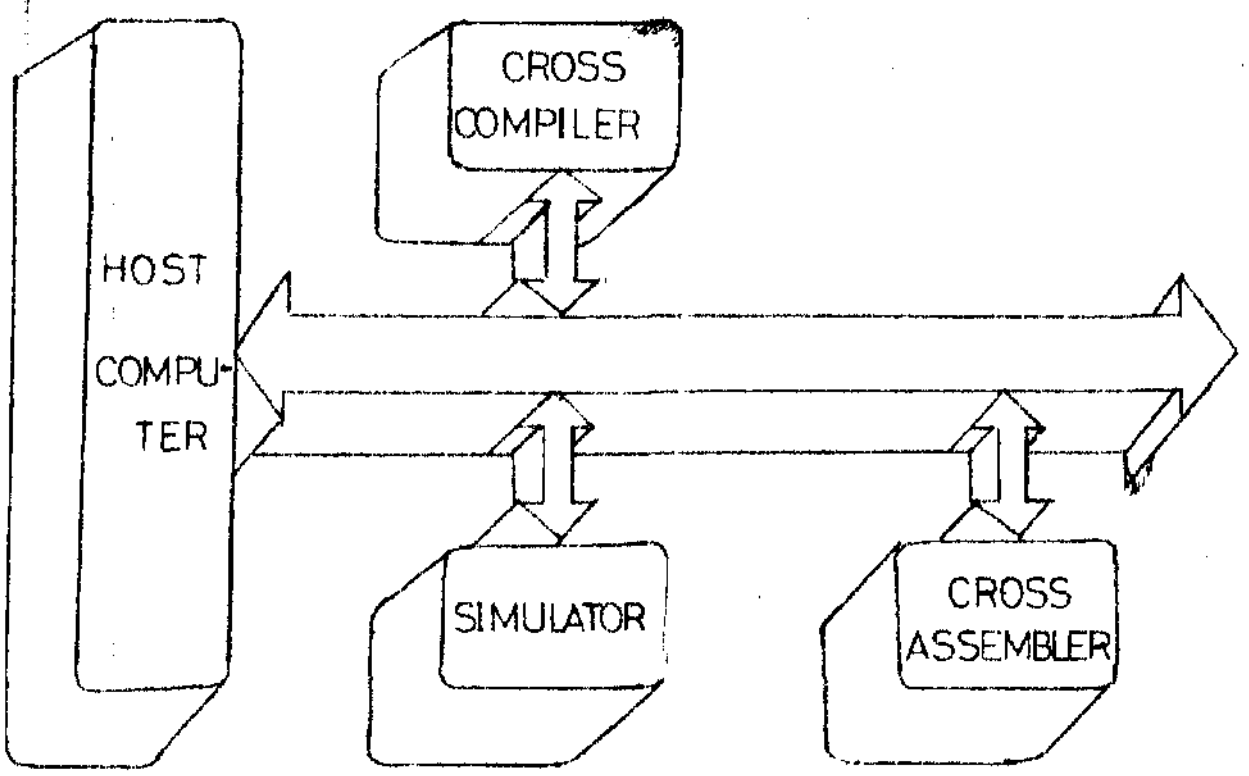


Figure 5.1 - Proposed Cross-Developement System

This thesis has developed an awareness and established a direction for microcomputer software development at the Ahmadu Bello University, and is submitted as a contribution to software product development in developing countries.

REFERENCES

The following abbreviations have been used:

CACM - Communications of the Association for Computing Machinery

Comput.J. - Computer Journal

E & P - Electronics and Power

JACM - Journal of the Association for Computing Machinery

Proc. IEE - Proceedings of the Institution of Electrical Engineers

Proc. IEEE - Proceedings of the Institute of Electrical and Electronic Engineers.

1. Aho, A.V. (ed), 'Currents in the Theory of Computing', Prentice - Hall, Englewood Cliffs, N.J. 1973.

2. Aho, A.V. & Johnson, S.C., 'LR Parsing', ACM Computing Surveys, Vol. 6, No. 2., June 1974.

3. Aho, A.V., Johnson, S.C., & Ullman, J.D., 'Code Generation for Expressions with Common Subexpressions' JACM, Vol. 24, No. 1, Jan. 1977:

4. Aho, A.V. & Ullman, J.D. , 'Principles of Compiler Design', Addison-Wesley, Reading, Mass., 1977.

5. Aspinall, D. & Dagless, E.L., 'Introduction to Microprocessors', Pitman, London, 1977.

6. Banatre, J.P., Routeau, J.P., & Trilling, L., 'An Event-Driven Compiling Technique' CACM, Vol. 22; No. 1, Jan. 1979.

7. Barker, H.A., 'The Microprocessor in Control', Proc. IEE, Vol. 126, No. 1, Jan. 1979.

8. Bass, C. & Brown, D. 'A Perspective on Microcomputer Software', Proc. IEEE, Vol, 64 No. 6, June 1979.

9. Bates, D. (ed), 'Structured Programming' Infotech International, Maidenhead, 1979.

10. Bauer, F.L., (ed), 'Software Engineering - An Advanced Course', Springer-verlag, Berlin, 1975.

11. Bauer, F.L. & Eickel, J. (ed), 'Compiler Construction - An Advanced Course', Springer-Verlag, Berlin, 1976

12. Bernhard, R., 'Computers: emphasis on software', IEEE Spectrum, Vol. 17, No. 1 Jan, 1980.

13. Blazie, D.B., & Levy, L.S., 'A Cross-Compiler for Pocket Calculators,', Comput J., Vol. 20, No.3. August 1977.

14. Bornat, R., 'Understanding and Writing Compilers Macmillan, London, 1979.

15. Brainerd, W. (ed), 'Fortran 77', CACM, Vol. 21, No. 10, Oct. 1978.

16. Burgess, C.J., 'Compile-time Error Diagnostics in Syntax-Directed Compilers', Comput. J., Vol. 15, No. 4, Nov. 1972.

17. Capece, R.P. & Posa, J.G., 'Microprocessors and Microcomputers-Technology Update', Electronics, Vol. 51, No. 22, 1978.

18. Caudill, P. 'Using Assembly Coding to Optimize HighLevel Language Programs' Electronics, Vol. 52, No. 3, 1979.

19. Cleaveland, J.C. & Satten, C.D., 'The Design and Implementation of a Simple Programming Language for Microcomputers' UCLA Quarterly.

20. Dahl, O.-J., Dijkstra, E.W. & Hoare, C.A.R., 'Structured Programming,' Academic Press, London, 1972

21. Donovan, J.J., 'System Programming' McGraw-Hill Kogakusha, Tokyo, 1972.

22. Feldman, J.A., 'Programming Languages, 'Scientific American, Vol. 241, No.6, Dec. 1979.

23. Feldman, J.A. & Gries, D., 'Translator writing Systems', CACM, Vol. 11, No. 2, Feb. 1968.

24. Foster, J.M., 'Automatic Syntactic Analysis' Macdonald, London, 1970

25. Frisch, M.J. & Liddiard, L.A., 'MNF Reference Manual', University Computer Centre, University of Minnesota , Minneapolis, Minn., 1976.

26. Gear, C.W., 'Computer Organization and Programming', McGraw-Hill Kogakusha, Tokyo, 1974.

27. Gilb, T., 'Present and Future Application Areas for Microcomputers' Management Datamatics, Vol. 4, No. 5, May 1975.

28. Graham R.M., 'Principles of Systems Programming', John Wiley & Sons, New York, 1975

29. Gries, D., 'Compiler Construction for Digital Computers' John Wiley & Sons New York, 1971.

30. Gries D., 'Use of Transition Matrices in Compiling', CACM. Vol, 11, No. 1, Jan. 1968.

31. Halstead, M.H., 'A Laboratory Manual for Compiler and Operating System Implementation', American Elsevier, New York, 1974

32. Higman, B., 'A Comparative Study of Programming Languages', Macdonald, London, 1967.

33. Hopgood, F.R.A., 'Compiling Techniques' Macdonald, London, 1969

34. Hsiao, D.K., 'Systems Programming' Addison-Wesley, Reading, Mass., 1975.

35. Hughes, P., 'Factoring in Software Costs Avoids Red Ink in Microprocessor Projects', Electronics, Vol. 51, No. 13, 1978.

36. Huskey, H.D., 'Compiling Techniques for Algebraic Expressions', Comput. J., Vol. 4, No. 1, Feb. 1961.

37. Jensen, K. & Wirth, N., 'PASCAL - User Manual and Report', Springer-Verlag, Berlin, 1975.

38. Kernighan, B.W. & Plauger, P.J., 'The Elements of Programming Style' Bell Telephone Laboratories, Murray Hill, N.J., 1974.

39. Lattin, W.W. & Bouvo, J.M. (ed) 'Microprocessor Applications', Proc. IEEE, Vol. 66, No. 2. Feb. 1978

40. Leventhal, L.A., 'Introduction to Microprocessors: Software, Hardware and Programming', Prentice-Hall, Englewood Cliffs, N.J., 1978.

41. Leventhal, L., 'The Advantage of 8080 and 6800 Data Manipulation Capabilities', Electronic Design, Vol. 25, No. 6, 1977.

42. Lin, Y.C. (ed), 'Microprocessors: Fundamentals and Applications', IEEE Press, New York, 1977.

43. McKeeman, W.M., Horning, J.J. & Wortman, D.B. 'A Compiler Generator', Prentice-Hall, Englewood Cliffs N.J., 1970

44. Motorola Semiconductor Products Inc., 'M6800 Microprocessor Programming Manual', Motorola Inc., Phoenix, Ariz., 1976

45. Mueller, R.A. & Johnson, G.R., 'A Generator for Microprocessor Assemblers and Simulators', Proc. IEEE, Vol. 64, No. 6, June 1976

46. Myers, R., 'The Need for Software Engineering', Computer, Vol. 11, No. 2, Feb. 1978

47. Palmer, P., 'Grammar', Penguin Books Harmondsworth, Middx., 1971

48. Parasuraman, B., 'High-Performance Microprocessor Architectures', Proc. IEEE, Vol. 64, No. 6, June 1976.

49. Pollack, S.V. (ed), 'Compiler Techniques', Auerbach, Princeton, N.J., 1972.

50. Posa, J.G., 'Pascal becomes software superstar', Electronics, Vol. 51, No. 21, 1978.

51. Posa, J.G., 'Programming Microcomputer Systems with High-Level Languages', Electronics, Vol. 52, No.2. Jan.1979.

52. Robinson, B.K. & Torsun, I.S., 'An Empirical Analysis of FORTRAN Programs', Comput. J., Vol. 19, No.1, Feb. 1976.

53. Rohl, J.S., 'An Introduction to Compiler Writing', Macdonald and Jane's, London, 1975.

54. Rohl, J.S. & Ellis, J.A., 'A Note on Compiling Arithmetic Expressions', Comput J., Vol. 15, No. 1, Feb. 1972.

55. Rosen, S. (ed), 'Programming Systems and Languages', McGraw-Hill, New York, 1967

56. Sebern, M.J., 'A Minicomputer-Compatible Microcomputer System: the DEC LSI-11', Proc. IEEE, Vol. 64, No. 6, June 1976.

57. Sippl, C.J., 'Microcomputer Handbook', Petrocelli - Charter, New York, 1977.

58. Smith, K.L.A., 'Approach to the Evaluation of Microprocessor Development Systems', E & M, Vol. 25, No. 3, Mar. 1979.

59. Soucek, B., 'Microprocessors and Microcomputers', John Wiley & Sons, New York, 1976.

60. Tanenbaum, A.S., 'A General-Purpose Macro-Processor as a Poor Man's Compiler-Compiler', IEEE Transations on Software Engineering, Vol. SE-2, No. 2, June 1976.

61. Watson, I.M., 'Comparison of Commercially Available Software Tools for Microprocessor Programming', Proc. IEE, Vol. 64, No. 6, June 1976.

62. Weingarten, F.W. 'Translation of Computer Languages', Holden-Day, San Francisco, CA, 1973.

63. Weissberger, A.J. & Keyling, G. Jr., 'Microprocessor Components and Systems', National Semiconductor Corporation.

64. Weller, J.J., 'Assembly Level Programming for Small Computers', Lexington Books, Lexington, Mass., 1975.

65. Wulf, W., Johnson R.K. Weinstock, C.B., Hobbs, S.O. & Geschke, C.M., 'The Design of an Optimizing Compiler', American Elsevier, New York, 1975.

66. Yourdon, E., 'The Revolution in Software Development', Keith London Associates . Hitchin, 1979.

67. Yourdon, E., 'Techniques of Program Structure and Design', Prentice-Hall, Englewood Cliffs, N.J., 1975.

68. Zaks, R & Steingart, D., 'Microcomputer Programming', Sybex, Berkeley, CA, 1976.

69. Bogod, J.M., 'Computing in Developing Countries' BCS address at CAN First Annual International Conference, Lagos, 1980.

70. Longe, O., 'The Potentials of Computer for Development', Keynote address at CAN First Annual International Conference, Lagos, April 1980.

71. Owolabi, 'An International Approach to Informatics for Development' UNESCO address at CAN First Annual International Conference, Lagos, April, 1980.

72. Rateau, 'Informatics at the Service of Development and the Role of the Intergovernmental Bureau for Informatics', IBI address at CAN First Annual International Conference, Lagos, April 1980.

A.1 The M6800 Prototyping System

The M6800 Prototyping System at the Ahmadu Bello University Electrical Engineering Department is based on one MC6800 Microprocessor Unit. Static memory is provided by one 1024 x 8-bit ROM (MC6830AL) while dynamic memory is provided by five 128x8-bit RAMs (MCM6810AL). Extefnal Communication is effected through two Peripheral Interface Adapters (PIAs - MC6820) and one Asynchronors Communications Inteface Adapter (ACIA - MC6850). Peripherals are a Keyboard/Display for entering and debugging user programs, and an Audio Cassette used as backing storage. A Clock Generator (MC6871B) provides a 1 MHz clock rate. The system is shown in Figure A.1.

The M6800 systems was designed such that peripheral parts appear to the microprocessor as simply memory locations on the address and data bus (A1). They are completely programmable from the bus and their real time status is available through the bus. This architecture firstly simplified the inteface between the memory and peripheral parts, and secondly, it eliminated the use for I/O instructions.

The M6800 family requires only one power supply (+5 volts), and the output buffers are capable of driving standard TTL. This means the M6800 family can directly interface with standard TTL logic without the need for additional power supplies.

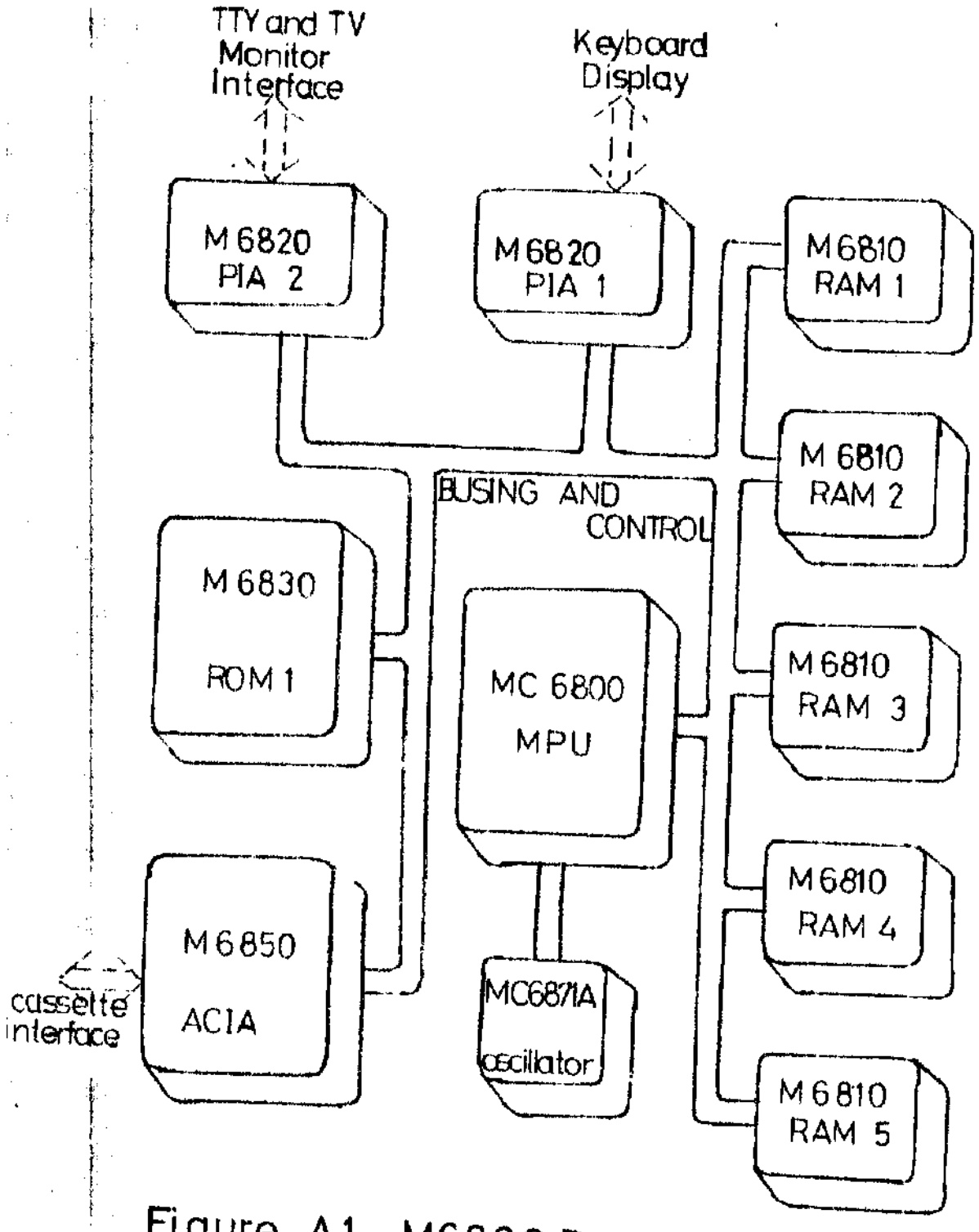


Figure A,1 - M6800 Prototyping System

The Prototyping System is controlled by a monitor - JBUG - which is resident on the ROM. The Keyboard/Display, in conjunction with JBUG, provides a means of examining operation of the Microcomputer and entering and trouble-shooting programs. JBUG allows examination of memory and registers, Magnetic tape loads and dumps, stepping breakpoints, program execution, and escapes. (A'2) These functions are controlled by keys on the Keyboards

---

A1. M6800 Microcomputer system Design Data, Motorola Semiconductor products Inc., Phoenix, Ariz.

A2. M6800 MEK6800D2 Evalucation Kit II Manual, Motorola Semiconductor products Inc., Phoenix, Ariz.