A MACHINE-INDEPENDENT MICROPROGRAM DEVELOPMENT SYSTEM

THESIS

Submitted in Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

of Rhodes University

by

MICHAEL JOHN WARD

December 1986

ACKNOWLEDGEMENTS.

I express my sincere gratitude to my supervisor, Peter Clayton for his help and encouragement during the development of this thesis. His assistance with all aspects of the project is truely appreciated.

I thank my mother for her love and support and especially my late father, who persuaded me to tackle the degree but who is unfortunately unable to witness its completion.

I am deeply endepted to my fiancèe, Carry for her love and inspiration throughout the year.

To fellow students John, Karen, Pete and Glenn, thanks for helping to make this project a truely memorable and enjoyable one.

Finally I acknowledge the financial support of the Council for Scientific and Industrial Research and Rhodes University.

ABSTRACT

The aims of this project are twofold. They are firstly, to implement a microprogram development system that allows the programmer to write microcode for any microprogrammable machine, and secondly, to build a microprogrammable machine, incorporating the user friendliness of a simulator, while still providing the 'hands on' experience obtained from working with actual hardware.

Microprogram development involves a two stage process. The first step is to describe the target machine, using format descriptions and mnemonic-based template definitions. The second stage involves using the defined mnemonics to write the microcodes for the target machine. This includes an assembly phase to translate the mnemonics into the binary microinstructions.

Three main components constitute the microprogrammable machine. The Arithmetic and Logic Unit (ALU) is built using chips from Advanced Micro Devices' Am2900 bit-slice family, the action of the Microprogram Control Unit (MCU) is simulated by software running on an IBM Personal Computer, and a section of the IBM PC's main memory acts as the Control Store (CS) for the system. The ALU is built on a prototyping card that plugs into one of the slots on the IBM PC's mother board. A hardware simulator program, that produces the effect of the ALU, has also been developed.

A small assembly language has been developed using the system, to test the various functions of the system. A mini-assembler has also been written to facilitate assembly of the above language.

A group of honours students at Rhodes University tested the microprogram development system. Their ideas and suggestions have been tabulated in this report and some of them have been used to enhance the

system's performance.

The concept of allowing 'inline' microinstructions in the macroprogram is also investigated in this report and a method of implementing this is shown.

CONTENTS.

TABLE OF FIGURES.

OUTLINE.

This thesis, presented in two parts, describes SYSMW, a microprogram development system. Part 1 looks at the field of microprogramming in general and the underlying motivations for designing this type of system. SYSMW is discussed in terms of its broad principles and comparisons are drawn with other similar systems. Part 1 concludes with a critical assessment of the project and a discussion of possible research projects for which SYSMW may be used. Part 2 deals with the design details of the system. The software and the hardware are discussed and an example architecture is presented. Part 2 also includes a user's guide for the system. The appendices contain listings of the 'TURBO' Pascal [TUR] code, circuit diagrams, further data pertaining to the hardware, and the example architecture files.

PART 1

1 : Introduction

This section describes microprogramming systems in general. It includes a short history of microprogramming and discusses why microprogramming has become a plausible design option for the systems engineer. Two main areas of research in this field are identified, namely firmware engineering and vertical migration. Although this thesis is concerned almost exclusively with firmware engineering, a brief scenario of vertical migration is included for completeness. Section 1 goes on to introduce general microprogramming terms, including relevant background information concerning development systems. Finally there is an introduction to the design objectives of SYSMW.

2 : General Concepts and scope

Section 2 discusses the requirements of development systems and how most of these are met by SYSMW. The system's hardware independence is described against the background of table driven assemblers. The discussion covers the use of templates to describe the machine and the use of defined mnemonics to facilitate easy microprogramming. SYSMW is then discussed in the context of microprogrammable machines and simulators. Major differences between other systems and SYSMW are described with explanations of how these are used to SYSMW's advantage. The testing of the system by post graduate students is introduced along with the motivations for developing a test architecture. Finally the system's ability to facilitate the use of 'inline' microcode is discussed.

3 : Comparisons with other Systems

This section provides a general comparison of SYSMW with other development systems. The comparisons fall into two areas, namely microprogramming languages and simulators. Included in these categories are points on how other microprogrammable machines compare. A separate section on AMD's SYS 29 seems justified as it appears to be the most widely used commercial microprogram development system [DAT]. In addition there is a survey of bit slice chips other than those used in the design of this system and a discussion of recent microprogramming developments.

4 : Discussion and Conclusion

Section 4 forms the overall project evaluation. Possible research projects using this kind of system are suggested.

PART 2

1 : Design Details

This section deals with the overall design specifications of the system. Micro1 and Micro2, involved in the first and second phases of microprogram development respectively, are described. Schematic diagrams and algorithms are included to aid the description. Micro1 and Micro2 are then drawn together, showing how they form a machine-independent microprogram development system. The thoughts and ideas acquired from the Computer Science Honours class at Rhodes University, after they had tested the system, are tabulated. It is pointed out which of these ideas were implemented as updates to the system and which were omitted, along with the justifications for these decisions.

Section 1 continues with a description of SYSMW's microprogrammable machine. The arithmetic and logic unit (ALU) hardware and the microprogram control unit (MCU) software are explained. Each of these parts is then discussed in more detail, including a section on how they communicate with each other. A simulator, that produces the effect of the ALU hardware, is also described.

2 : Example

This section takes the form of an example of the use of the microprogramming system. The implementation of a simple assembly language is shown. The concept of 'inline' microcode is discussed in terms of a working example. General comments on how the assembly language operates and a brief discussion of the mini-assembler program, a code generator for the above language, are also included.

3 : A User's Guide

Section 3 is a general user's guide which includes specific operating instructions and comments on the help files and the error messages, for all programs in the system.

Appendices : A to H.

Appendices A to D contain the 'TURBO' Pascal program listings. Appendix E lists the help files for all the programs while appendices F through H contain the micro-order tables, the example file listings, and the hardware circuit diagrams, respectively.

PART 1

1 : INTRODUCTION.

1.1 : Introduction to Microprogramming.

There are two types of computers available today, those with 'monolithic' and those with 'bit-sliced' based processors [SKO]. Monolithic machines have a fixed instruction set and fall into the class of conventional assembly language machines. Bit-slice machines on the other hand can have very different language instruction sets chosen by the designer or user.

The two types of machines do however have a lot in common. Figure 1.1 shows the constituent functional units of a typical computer, namely the memory, the central processing unit (CPU), and the input/output (I/O) drivers.

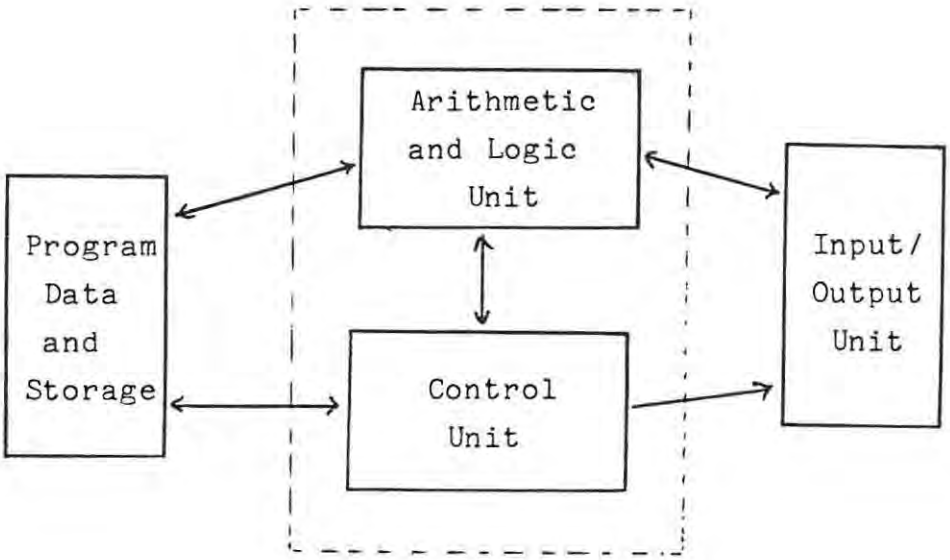


Figure 1.1
SCHEMATIC DIAGRAM OF A GENERALISED COMPUTER

As a consequence of having two design philosophies, microprocessor technology has evolved in two distinct directions [MCG]. The first is the design of microprocessors that have more and more of the features of large computers. The second is the design of microprocessors more closely adapted to the digital hardware or specific application. The former class is dominated by 16-bit microprocessors while the latter is characterised by the development and use of bit-slice processors [MCG].

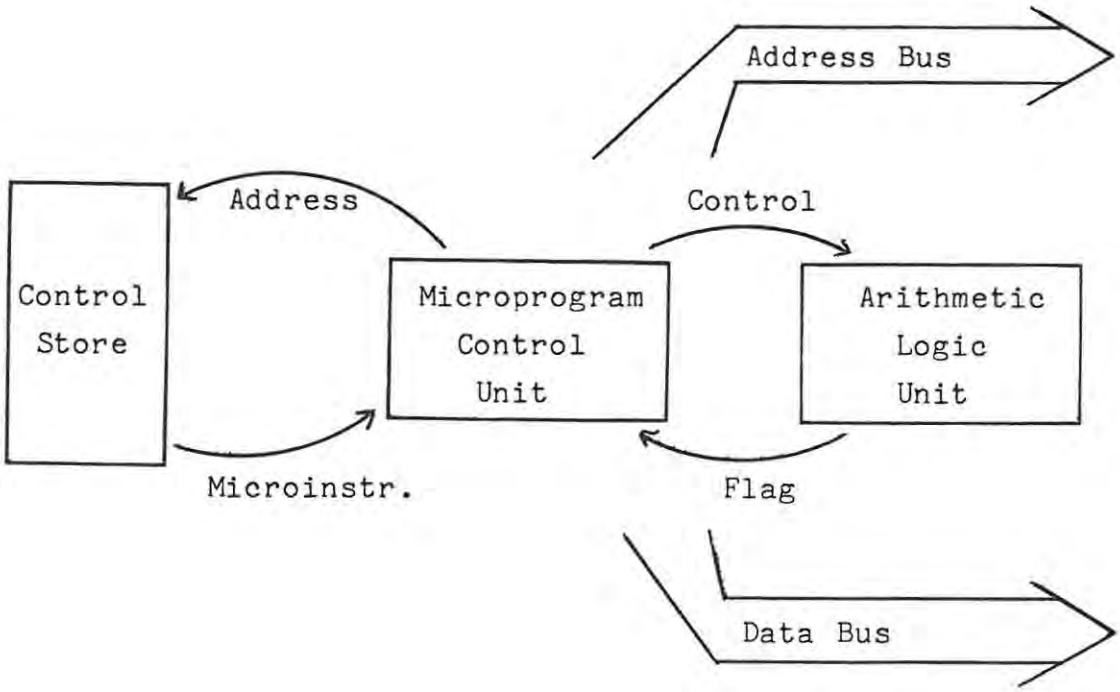


Figure 1.2
SCHEMATIC DIAGRAM OF A MICROPROGRAMMABLE CPU

In microprogrammable machines, the CPU is the only part of the computer that differs in design from conventional, fixed instruction set machines. A microprogrammed CPU consists of three parts, the arithmetic and logic unit (ALU), the microprogram control unit (MCU), and the

control store (CS) [COL]. Conventional machines employ hard-wired sequential logic to perform the functions of the MCU and the CS and these together with the ALU are usually all on a single chip. (In bit-slice machines, the necessary control is performed by the MCU and the CS.) The permanent nature of the hard-wired control section gives rise to the term 'fixed instruction set'. The relationship between the ALU, MCU and CS for a microprogrammable processor is shown in Figure 1.2.

1.2 : History and Perspective.

In 1951, wanting to improve the reliability by increasing the regularity of the control unit in CPU's, Maurice V Wilkes suggested the idea of having a microprogramming level below the machine language level [WIL]. He intended the microprogram to serve as an alternative method for the design of the control unit, as opposed to hard wired designs used exclusively at the time in the fixed instruction set processors. His primary motive was a systematic design method as a replacement for the random logic of the control unit. This idea is preserved in its pure form in horizontally microprogrammed machines [CHR].

The IBM System/360 was one of the first computers to use microprogramming [MCG], and today most main frame computers, and numerous minicomputers, have a microprogrammed level beneath their machine architectures [CHR]. This statement is perhaps a bit nebulous as one should not refer to microprogramming in the binary sense but rather the degree to which a machine is microprogrammed [MYE]. In many applications the microinstruction signals are decoded before being fed to the hardware and the control system is therefore a combination of hard-wired and microprogrammed control.

The term 'firmware' was coined by Asher Opler when he projected an idea of what fourth generation computers would look like [OPL]. He felt that microprogramming would lead to the wide spread use of no-order-set/no-data-structure computers, which could be individually tailored for specific applications through the use of replaceable microprograms. The microprograms would either be available from the manufacturer, or they could be prepared by the users themselves. He also foresaw the inclusion of high level functions into the firmware [OPL]. This has become known as vertical migration.

The microprogramming concept has no clear-cut formal definition, although Myers suggests the following working definition [MYE1].

> "Microprogramming is the process of producing microprograms. A microprogram is a form of stored-program logic that explicitly and directly controls the major logic devices of digital systems (e.g., registers, ALUs, counters, busses, memory). As such, a microprogram is a substitute for a sequential logic control network. The most common application of a microprogram, but certainly not the only application, is to give a processor a particular instruction-set architecture."

If a monolithic processor and its support chips can be used in a particular application, then the tasks of digital systems design and physical layout are substantially reduced. However this is not a viable solution to all design problems as monolithic microprocessors are relatively slow, and as has been seen, have static instruction sets. More flexible fourth generation building blocks are sometimes needed and bit-slice devices are an answer to this requirement [MYE1].

The microprogramming option is usually selected by the design engineer because it improves flexibility, performance and LSI utilization of

microprocessor systems [MIC]. Microprogramming also leads to a more structured organization while cost and design time are reduced. Diagnostics can be implemented easily, design changes are simple, field updates are easy, adaptations are straight forward, the system definition can be expanded to include new functions, documentation and service are easier, and design aids are usually available [MIC].

Interest in and the use of microprogramming has soared, driven by recent advances in technology [SCH]. This renewed and widespread interest has been facilitated by the availability of high speed memory and powerful LSI bit-slice processor components [MEZ3]. Microprogramming is being used to reduce the complexity of VLSI design because previously complex hardwired logic can become impossible to lay out on a single chip [SCH]. The recent push towards fifth generation computers has resulted in microprogramming being used to migrate functions into intelligent components, and cheap hardware has caused more architects to use the parallelism available in horizontal microcode to speed their systems up [SCH]. The direct benefits of microprogramming are thus a reduction in the design time of the control section, and increased performance of the system [MEZ3].

There are of course disadvantages in using microprogrammed control as opposed hard-wired control. Myers puts forward a few of these, namely, that it is more costly for simple machines, it is sometimes slower, the designs are easier to copy, and the user is tempted to make microprogram changes [MYE1].

The graph (figure 1.3) shows how the cost involved in developing the system compares to the richness of the resulting instruction set for sequential-logic and microprogrammed controlled processors [MYE1].

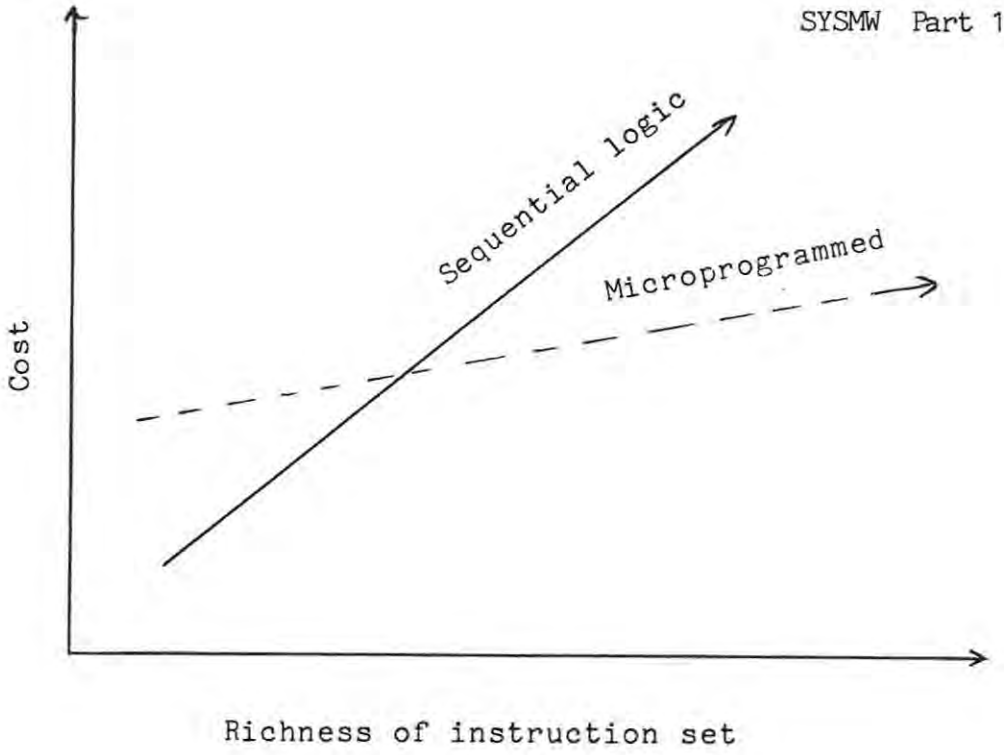


Figure 1.3
GRAPH OF RICHNESS OF INSTRUCTION SET REPERTOIRE vs COST

There are a few further factors which influence the decision of whether to use a fixed instruction set processor or a microprogrammable bit-slice processor [WHI]. These include the physical size of the resulting board (approximately 10 times the number of chip are required for a bit-slice design than for a fixed instruction set machine), the word length (bit-slice offers any multiple of 4 bits - more recently bit-slices have become available in sizes from 2 to 32 bits wide), and speed (bit-slice devices are about 10 times as fast). In general it can be said that bit-slices are applied in three basic areas, namely, machines with long words, machines with special instruction sets and high speed machines [WHI].

Microprogramming has over the years, branched into two main areas of research. The first of these areas addresses the question of how to write microprograms. This has become known as 'firmware engineering'.

The second area considers how to isolate functions which are candidates to be migrated into the firmware. This has become known as 'vertical migration' [MON].

The growing interest in firmware engineering is a consequence of the availability of user microprogrammable machines, and as long as only machine instructions are emulated, microprograms are relatively simple [MON]. This thesis is concerned almost exclusively with this branch of microprogramming research.

1.3 : Vertical Migration.

The classical form of migration is the migration of functions from software to firmware [STA]. A generalisation of the classical form of migration is called vertical migration which is the migration of functions from a high level abstract machine to a lower level abstract machine. Hence vertical migration includes software migrations as well as the software to firmware migrations [STA].

If there is a desire to migrate functions with complex interfaces into the firmware, the microprograms are no longer simple. Moving the function down in the hierarchical structure allows for the faster execution of this function, but at the same time makes its implementation more difficult and later changes more costly. The main driving force to move functions from software into firmware is the gain in performance. Vertical migration is almost exclusively understood in this sense [MON].

To perform the migration, a systems programmer must understand the function to be moved, and must know the 'high-use' paths of that function, as well as all the control and data interconnections related

to the function [STA]. The performance improvements attainable by vertical migration arise in two ways. Firstly, the function is moved to a faster level, and secondly, higher level functions are often general enough to handle a variety of inputs. By making the function less general it can be optimised [STA].

Grossman et al. show that the performance of microprogrammable computer systems can be greatly improved by microcoding frequently executed, and CPU intensive, software functions. These authors quote percentage benchtime improvements for vertical migration techniques implemented on a Burroughs B 1726 [GRO].

'Today it is a well known fact that the efficiency of solving a particular problem (executing a problem-oriented program) depends primarily on the degree to which the architecture of the computer supports the problem solving primitives. A measure of the difference between the concepts in the programming languages and the concepts in the computer architecture is the "semantic gap"' [LUQ]. In most present day systems this gap is large. Vertical migration can close the gap by moving selected functions to a more optimal level in the hierarchy and hence improve the system performance. One should realise however that microcoding an inefficient algorithm, may result in faster execution than with a software implementation, but the algorithm will still remain inefficient. A formal statement to this effect has been coined as Raucher's Law [RAU]. With the gains in being able to close the semantic gap, it seems obvious that microprogramming is going to become more and more important in the future, as the call for more specialised and faster machines increases.

1.4 : Microprogramming Terminology.

Support tools for microprogramming are either hardware based or software based. The software supports for microprogramming can be subdivided into two distinct classes namely, microprogramming languages and simulators [MEZ2]. The microprogramming languages range from simple microprogram assembler languages, through register transfer languages, to procedural oriented machine dependent microprogramming languages, and beyond [RAU]. Simulators satisfy two different requirements in that they can be used for the verification of the hardware and for microprogram checking [MEZ2].

Within the field of simulators it is interesting to note that there is a distinct difference between simulation and emulation. The term simulation is used when the software program 'acts' like the prototype processor but it runs slower and therefore critical timing relationships are impossible to verify [EDE]. Emulation involves the construction of a hardware model of the prototype processor. It may be constructed using discrete logic or another type of microprocessor [EDE]. An emulator therefore allows one to check the critical timing between the software and the prototype hardware. The term emulator has been used loosely in this report and does not include the constraint on critical timing.

The term emulator can also be used to describe 'a complete set of microprograms which, when imbedded in control store, define a machine' [LEW]. The machine which is realised by the emulator is called a virtual machine, and the machine which supports the microprograms, a host machine [LEW]. In other words the microcode emulates the higher level macroinstructions that define a virtual machine.

The field of microprogramming languages gives rise to a multitude of terms and definitions. Each primitive function at the microprogramming

level is referred to as a microoperation or micro-order. One or more micro-orders make up a single microinstruction [CLA1] which performs one micro-operation [KRA] in each micro clock cycle. These microinstructions are stored as words usually in a high speed memory area, called the control store [CLA1].

Macroinstructions are strings of Ø's and 1's that reside in main memory and a sequence of macroinstructions is termed a macro-program or main program [KRA]. Each of these macroinstructions is emulated by series of microinstructions collectively called a microroutine. One or more microroutines form a microprogram [CLA1].

The microinstructions may either be wide (anything above about 6Ø bits wide), or narrow (anything below about 3Ø bits wide). The former is termed horizontal microprogram design and the latter vertical microprogram design [CLA1]. In horizontal microprogramming, one bit of the microinstruction corresponds to one command [OBR] or one physical control signal. Some space is usually wasted, but this method is useful from a topological point of view, is easy to implement, and is usually faster than a vertical design [OBR]. Horizontal designs allow the microprogrammer to explore the principles of concurrent processes as it is possible to execute various functions at the same time.

In vertical microprogramming, the microinstructions are encoded and the microprogrammer becomes more divorced from the actual logical devices in the system [CLA1]. The benefit of shorter microinstructions is offset by the added cost of decoding logic [OBR]. In the limit, the microinstructions become macroinstructions and the control system is hard-wired as in the case of a fixed instruction set machine. Any designs that fall into the grey area between vertical and horizontal, are usually referred to as diagonal microprogram designs [CLA1].

A further extension of microprogramming, called nanoprogramming, is

used in the machine Nanodata QM-1 [KRU]. In nanoprogramming, the microinstructions address a lower level nanostore which contains the nanoinstructions. These nanoinstructions are effectively subroutines called by the microinstructions.

This thesis is concerned mainly with horizontal microprogram designs as these are considered true microprogramming from an academic point of view. The philosophy behind horizontal microprogramming is also consistent with the original ideas of Maurice Wilkes.

1.5 : Microprogram Development Systems.

Microprogramming as an end-user tool has evolved slowly due to three major obstacles that have stood in its way [VAX]. Firstly, there is a need for very fast random access memories. Secondly, microprogramming requires special knowledge and it has taken time for the subject to be incorporated into curricula and be taught. Thirdly, and most importantly, the expansion of microprogramming is the inclusion of generality and extensibility in computer design. A system with no address space for user control programs or no built in features to support general microprogramming, makes writing user microcode extremely difficult [VAX]. This means that software microprogramming support tools have to be developed.

In general a development system should have all the facilities necessary for the development and testing of microprograms. These include microcode definition, assembly of the definitions into binary microinstructions, verification of the hardware, testing and debugging of the microcode routines, and loading of the microcode into the control store memory area.

Corcoran has the following feeling on development systems: 'The availability of a higher level language programming system which can be systematically and easily tailored to a specific target machine architecture without having to rewrite any part of the compiler software, would serve to bring a level of standardisation into this field and could also increase the number of users who would consider using microcoded architectures' [COR].

The main problem is that, although bit-slice processors offer a considerable advantage in execution speed and processing capability, it is generally hard to get them operational. The problems include the complexity of the microinstructions and the intimate relationship between them and the hardware for which they are intended [COL].

Tsuchiya also expresses a need for better and more systematic development procedures and methodologies, because of the increased interest in firmware [TSU]. Firmware developers usually turn to software disciplines for directions and while software and firmware are inherently different, they do share many common characteristics. If viewed in this light, microprogramming is simply an adaptation of programming techniques to implement the hardware control sequencing [TSU]. This is however a very simplistic view, which becomes less and less pertinent as one moves away from vertical towards horizontal microprogram designs.

There has recently been a great deal of interest in the effective use of high-level microprogramming languages [MUE]. The primary goals behind their use are the reduction in development time and the increase in reliability, without the loss of efficiency, in the end product. Achieving these goals is largely dependent on the translator which maps high level descriptions of the desired machine to a control store implementation of that machine on a microprogrammable host.

The main issues which help to distinguish between different microprogramming languages are increased machine independence, efficient code generation and structured programming [BAL]. These often cause conflicts in the language design necessitating compromises. The state of development of microprogramming languages is nowhere near the level of high-level programming languages, mainly due to the efficiency of the microcode to be generated. Balakrishnan et al. [BAL] list major deviations from the conventional languages due to the following characteristics of the microprogramming environment: the support of parallel operations as opposed to serial, optimality considerations at a low hardware level, and the need to support a wide range of architectures.

There are two main approaches to the generation of microcode from a high level source language. These are, firstly the type where a separate translator has to be written for each machine and secondly, the type where there is a single translator, but the microarchitecture has to be described for each different machine [BAL].

Support tools for firmware engineering have lagged behind other developments in microprogramming because microprograms have up until recently been relatively small in size [LEW]. This is a problem because microprograms are now becoming very large, as VLSI structures become more available, and as operating system programs migrate into the control store [LEW].

The increase in microprogram size makes the task of debugging the microcode a formidable one [MYE2]. This has fostered a need for simulators to facilitate the debugging of microcode [MYE2]. Simulators are useful as they allow the user to monitor a completely controlled execution of the microinstructions. Single stepping through the microprogram is easily implemented and the values of various flags and registers can be traced.

1.6 : The Design Objectives of SYSMW.

Microprogramming is one of the few areas in Computer Science that overlaps naturally with Electronic Engineering and it is one that can give the student some contact with hardware design [CLA3]. Myers specifically states that microprogramming requires both engineering and programming knowledge [MYE1]. It is very important that the students be able to verify their designs by executing a few microprogram segments on readily available equipment [CLA3]. One of the primary design objectives of the microprogram development system, SYSMW, described in this report, is that of providing a system on which the practical part of a graduate course in microprogramming may be run.

Microprogramming has become the means of implementing the machine language instructions of a conventional computer. Vertical migration has caused functions to be moved from the software to the firmware. These and other factors tend to increase the complexity and volume of the firmware and there is therefore an overall increased importance of microprogramming and firmware design. This makes it highly desirable to have courses pertaining to these areas in a modern computer science curriculum [BEH]. A course of this nature could perhaps be divided into two distinct parts. Firstly, a firmware design section where students learn to specify the behavior and performance of microprograms in an operational specification method. Secondly, a microprogramming section where the student gains 'hands on' experience in the microprogramming of a particular computer by writing microcode segments in a microassembler language and by loading and testing and debugging them [BEH]. SYSMW attempts to cover both of these areas, with perhaps more emphasis being placed on the latter.

There are two options available to the designer of a microprogram development system. Either a universal development system can be

constructed, which is usually very expensive, or a cheaper and more efficient system relatively dependent on the application site can be developed [MON]. SYSMW combines the generality of a universal development system with the low cost and efficiency of a site dependent system.

A necessary condition for a machine-independent code generation system is that the specific target machine and microcode specifications are independent of the proof system [MUE].

A machine-independent microprogramming language is not an adequate substitute for a machine-oriented language if optimally efficient microprograms are required. This is the view of Sommerville [SOM] when he identifies the following four application areas for machine-independent microprogramming systems: (1) as a machine-description language formally defining a machine architecture, (2) as a research tool allowing research workers to produce and evaluate experimental machine architectures quickly, (3) as a portability tool which can be used to emulate a particular architecture on another machine, and (4) as a tool for the systems programmer, allowing him to extend a machine instruction set for his own application. SYSMW is capable of being used in all four of these areas, but points 1, 2 and 4 are of particular interest as it is for these reasons that SYSMW was developed.

Renyi et al. [REN] feel that it is of the utmost importance to have an easily manageable and versatile hardware-software interface, during the development phase of a microprogrammed system. A user oriented microprogramming system ought to have an easy to use, reliable and pleasant interface [BAL]. SYSMW was developed with ease of use and versatility as important criteria.

The basic design approach for SYSMW was to tackle the topic from an Electronic Engineer's point of view rather than form a Computer

Scientist's point of view. Most other proposed microprogram development systems [BAB] [SOM] [COR] [SKO] [BAL1] [CHA] [MEZ1] [BAL2] [SRI] approach the problem from the software side and try to match high level constructs with the very low level, parallel type, microcode application. The SYSMW philosophy is different, providing the user with a new and divergent angle on microprogramming. This approach assumes that the user already has a bit-slice machine that he now wishes to microprogram. He would initially like to be able to develop only a small subset of the target language, in order to test the system functions and validate the hardware design. He would perhaps have no aids other than the data sheets supplied with the chips. He should now be able to use the SYSMW microprogram development system to get the machine up and running. The data sheets accompanying the chips fit in perfectly with the system and provide the necessary starting point for the target machine definition.

Fletcher's text entitled 'An engineering approach to digital design', provides a good overview of the concepts of a programmable system controller, from an engineering point of view [FLE].

The SYSMW Triple-M microprogrammable machine may be used to emulate and extend existing architectures or to develop and evaluate new ones. This microprogrammable machine makes the SYSMW development system complete, as it provides the necessary tools to realise the objectives of microprogramming systems [SOM].

SYSMW also includes a simulator that produces the effect of the ALU hardware. Myers et al. recognise the general objectives of simulators as follows: being able to display the state of the machine, being able to modify the state (registers and memory), being able to initialise and record various states, being able to set breakpoints and maintain timing data where possible, to be of an interactive nature, and to take cognisance of human factors (the man-machine interface and user

friendliness) [MYE2].

Working with microprogrammable processors seems likely to provide a better understanding of the internal operations of programmed machines. Microprogrammable devices permit the user to experiment with different instruction sets and try out different architectures, within limits. If the user of a bit-slice processor could more easily experiment with the relationships between macroinstructions and microinstructions, and with the details of microprogram execution, he could better understand the principles involved in the underlying hardware [DAV].

2 : GENERAL CONCEPTS AND SCOPE.

2.1 : Requirements of Development Systems.

Horizontal microprogramming has become extremely attractive in most high speed and real time applications due to the progress in the design and the fabrication of bit-slice devices [MEZ1]. This growing interest in microprogramming has lead to a need for tools that facilitate the development and the debugging of microprogrammed systems. Microprogramming support tools fall into three main categories; (1) microassemblers that allow one to represent microcode in a symbolic language, (2) high level microprogramming languages and their associated compilers, and (3) simulator programs that simulate the data flow of a microprogram at the micro-operation level [MEZ1]. Each of these tools has general validity in that it can be designed to adapt and process any kind of microprogrammable system. SYSMW, the system described in this report falls into groups 1 and 3 of the above microprogramming support tools.

There is a need for software tools for microprogram development. In general, machine dependent tools do not turn out to be cost effective. The availability of machine-independent software aids is therefore imperative. Such systems should be both flexible, in that the tools must be usable for different designs and products, and modular, in that there should be a uniform design approach across the different tools that constitute the whole system [MEZ1].

There are two parts to the design of any digital system. Firstly one has to become familiarised with the basic components of the system, and secondly one has to develop the system with all the attendant hardware, firmware and software [DIM]. The following steps are relevant in the development process of a microprogrammed bit-slice system: [GIB]

1- do a preliminary investigation

2- construct a block diagram
3- decide on the bit-slice chips to be used
4- decide on the microword width
5- decide on the microword format
6- write the microcode
7- assemble the microcode
8- debug the microcode
9- load the microcode into ROM
10- layout and produce the printed circuit board
11- test the complete interface

The SYSMW development system aids the microprogrammer in the execution of steps five through eight.

The basic function of a microprogram development system can therefore be seen as providing a means of writing and debugging the microcode. Desirable features of such a development system include: the ability to readily modify a microword, the ability to vary the method of execution including single stepping, the ability to support a number of different bit-slice families, and the ability to allow for changes in the format (field definition) of the microword [GIB]. SYSMW is capable of supporting all these features.

The development of SYSMW can be divided into two distinct parts. The first part concerns the specification of the microcode. The programs involved here are Micro1 and Micro2, and together they form a machine independent microprogram development system. This system formed the basis of a microprogramming exercise for the Computer Science Honours class at Rhodes University. The second part describes a specific microprogrammable machine built to encorporate the ideas of a user friendly simulator while still providing the 'hands on' experience obtained from working with actual hardware components. An assembly language was developed to test the functions of the microprogram

development system and the Triple-M microprogrammable machine. A small assembler program was developed to facilitate assembly of this language into machine readable opcodes.

A detailed description of the system components and how they operate can be found in Part 2.

2.2 : SYSMW in the Context of Table Driven Assemblers.

The idea of basing the SYSMW microprogram development system on a table driven approach was obtained from a paper written by Clayton [CLA2]. The paper discusses the implementation of a table-driven code generator for converting intermediate codes (for a hypothetical machine) into machine code. The system consists of two definition programs that define the machine and define the code-skeleton. The third program is the general table-driven code generator [CLA2].

SYSMW employs two programs to produce its object microcode. The first of these, Micro1, is used to describe the target microprogrammable machine, and the second, Micro2, defines the microcodes and then assembles them into the binary microinstructions.

2.2.1 : Machine Description.

Given a microprogrammable machine, two basic operations must be performed before the microcode can be developed. Firstly, the microinstruction format must be described, and secondly, the machine architecture has to be described [MEZ1]. Micro1 handles both of these operations.

The first phase of machine description involves defining template formats for the various micro-orders in a single microinstruction. These templates divide the microinstruction into fields with each field capable of holding one micro-order. For example one would define a field to hold the control bits that instruct the ALU to perform a specific arithmetic or logical function. In the case of the SYSMW's microprogrammable machine (Triple-M) this field is three bits wide.

The second phase is concerned with defining the specific bit pattern values for each micro-order. In most cases there would be many possible functions that could use the same format template. Each of these different functions or micro-orders is given a unique name and with each name is associated a specific bit sequence. For the case of the ALU function, one would be able to define eight different functions for the three bit wide field. Each of these micro-order function definitions would use the same format template.

Below is shown how a three bit wide template could be defined in a microinstruction that is nine bits wide.

```
Format template name:   ALU function
Format number:          1
Template:               3x 3a 3x  (ie  xxxaaaxxx)
```

Now using this template one could define eight possible micro-code definitions for the possible ALU operations. One such definition is shown below.

```
Micro-order definition mnemonic: ADD
Binary value:                    000
Format number:                   1
```

The format template records and the microcode definition records can

then be combined in an assembly phase to set up a table file. The resulting entry in this file for the 'ADD' mnemonic using format number 1, is given below.

```
Micro-order definition mnemonic: ADD
Binary value:                    000
Template:                        3x 3a 3x
```

All the entries in the table file constitute the complete machine definition, as each entry contains the micro-order mnemonic, its associated binary value, and its precise position in the microinstruction.

SYSMW's Micro1 program leads the user through these stages of machine description in an interactive, user-friendly manner. The layout and content of the interactive session ties in very closely with the specification sheets that are supplied with the chips, making the machine description that much easier.

2.2.2 : Micro-code Development.

In comparing microcode to the code written in conventional high-level programming languages, two major differences appear to make microprogramming more difficult. The first of these, namely the low power of the microinstructions, arises out of the greater number of components to be considered in microprogramming. Here the limit to man's short term memory plays a major role. The second difference lies in the microcode's lack of features like block structures (and general lack of structure), and the consequent high degree of mutual influence between components [ZIN]. A microprogramming development system should take cognisance of these facts and attempt to alleviate the associated problems, as is done by SYSMW's Micro2 program.

The Micro2 program allows the user to develop the microinstructions using the micro-code mnemonics defined during the execution of Micro1. Instead of trying to write all the microcode at once, the development is divided up into small microcode segments. Each segment performs a specific function, for example, a boot routine or a routine to emulate one high level language instruction. In the case of emulating an assembly type language, a microcode sequence is associated with each macroinstruction opcode. For the fetch cycle (and other functions) that has no associated opcode, the microcode sequence is termed a microroutine.

This process of fragmentation simplifies the task of the microprogrammer. It allows the system to be built up in small steps, with the programmer adding more and more functions as the need arises. In this way the system may be tested with perhaps a small subset of the final architecture.

The number of microinstructions that can be associated with each function is not limited, allowing the development of microcode segments that are able to perform more high level language type functions. In general, the more complicated the function to implement, the longer will be the sequence of microinstructions necessary to emulate it.

Once the user has defined the microinstructions in symbolic form, they can be assembled into the binary representation for loading into the control store. SYSMW employs a two pass assembler to realise this function.

2.3 : SYSMW and Microprogrammable Machines and Simulators.

Microprogrammable machines are usually built using discrete bit-slice hardware components. These machines have associated high speed control store sections and most systems run at clock speeds of around 20MHz. The high speed and suitably tailored architecture enables the microinstructions to emulate the higher level instructions very efficiently. Hardware systems, other than development systems, do not have any debugging facilities and they assume that the microcode has been tested and is correct. Development systems, such as AMD's SYS 29, need large amounts of associated code to render the hardware sufficiently user friendly, so as to allow the development of the microcode to be done relatively easily.

At the other end of the scale are the simulator programs. These programs usually run on a host 'fixed instruction set' machine, and try to emulate the action of the bit-slice hardware as closely as possible. They are very user friendly and usually provide extensive features for the simple debugging and testing of the microcode. Simulators lose some credibility, however, in that they do not reflect the intricate timing constraints of a high speed processor and there is no guarantee that microcode developed on a simulator will run on the machine that it supposedly simulates. There is also no sense of having gained 'hands on' experience, as this can only be obtained by working with the actual hardware.

SYSMW's Triple-M machine combines the user friendliness of a simulator with the 'hands on' feel of a hardware machine. This was achieved by dividing the microprogrammable machine into two parts. Half is constructed with hardware components, while the other half is simulated by software running on a host machine. There is no gain in execution speed above that of a normal simulator, as the hardware is mostly waiting for the simulator software to catch up. This dual nature of the

Triple-M machine forms the fundamental difference between it and other microprogrammable machines.

Triple-M's arithmetic and logic unit (ALU) is constructed using components from AMD's Am2900 bit-slice family. The microprogram control unit (MCU) and control store (CS) are simulated by 'TURBO' Pascal code running on an IBM Personal Computer.

All microprogrammable machines have two types of memory. There is a high speed control store that holds the microinstructions and there is the conventional main memory that stores the user macroprograms. The main memory is usually slower and is the same width as the data path for that machine. The control store is the same width as the microinstructions and is normally non volatile. The Triple-M machine uses the same IBM memory to simulate the main macroprogram memory and the control store. This effectively means that the control store is a lot slower that the high-speed memory that is normally used, and it also means that the microprogram is easily changed as it resides in volatile random access memory.

2.4 : SYSMW testing and Inline Microcode.

The basic functions of SYSMW were tested by implementing a machine language type architecture. This proved very satisfactory as the microcode segments for each of the macroinstructions were of a manageable size. Routines were also developed to test the operation of the macro- and micro-status register flags. A detailed discussion of the example language is given in Part 2.

The Micro1 and Micro2 programs, forming a machine-independent microprogram development system, were used as the basis for an exercise

in a post graduate course in microprogramming. The two programs were used to develop microcode segments for a hypothetical machine. This proved extremely useful as the author was able to gauge the success and usability of the system. Many of the helpful points raised by the students were used to upgrade the system and correct any bugs that became apparent. Further details regarding this exercise are discussed in Part 2.

The 'TURBO' Pascal language supports an interesting feature that allows machine code instructions to be inserted directly into the program text. The reserved word 'inline' is simply followed by one or more code elements separated by slashes and enclosed in parentheses [TUR]. This concept lead to the idea of having a microprogrammable system that allowed inline microinstruction to be executed from within the emulated macroprogram. In the example of the machine language type architecture, this means that the assembler should allow the use of inline microinstructions within the body of an assembly language program.

Inline microinstructions are a useful means of extending the existing capabilities of the language, without having to alter the microprogram in the control store. In the case where the microprogram resides in a non-volatile ROM, this may be the only means of being able to develop and execute ones own microinstructions. It is also useful for experimenting with 'one-off' microinstructions, that if proved useful, could later be incorporated in an updated version of the microprogram. The one important constraint is that of execution speed. A wide microinstruction would take up several consecutive locations in main memory, and it would take several clock cycles to build up the microinstruction in a temporary latch, before it could be executed.

An alternative method for implementing inline microinstructions might be to include a few residual-control microinstruction registers. At the start of the macroprogram one could then define all the inline

microinstructions that are to be used within the body of the program. The microinstructions would be loaded into the residual control registers at the start of execution, to be used later in the program. The number of inline microinstructions that could be defined in this way is limited by the number of residual control registers made available. The advantage here is that the microinstructions only have to be loaded in once and thereafter they can be used several times from the residual control registers without any loss in execution speed.

SYSMW's Triple-M machine uses the former approach to implement the inline microinstructions and further details on the implementation can be found in Part 2.

3 : COMPARISONS WITH OTHER SYSTEMS.

3.1 : Microprogramming languages and Simulators.

The two distinct classes of microprogramming software support tools are microprogramming languages and simulators [MEZ3]. Today many microprogramming languages are available, varying from the usual microassembler languages, which express absolute microcode by means of mnemonic and symbolic representations, to the so called 'high level languages', which allow the user to write microprograms in a conventional and sequential and procedural fashion. Microprogram simulators are a completely different kind of software support, used mainly for microcode verification. In most cases, the simulators are machine specific, in order to increase their efficiency [MEZ3]. The paragraphs that follow discuss microprogramming languages and then proceed to describe various simulator packages. Where appropriate, comparisons are drawn between the features of the other systems and those of SYSMW.

3.1.1 : Microprogramming Languages.

The MPG system [BAB] consists of a high level microprogramming language called MPGL, and a processing system called MPG. The microprogrammer uses MPGL to write microprograms sequentially in a machine-independent fashion, while MPG provides the facility to translate these microprograms into efficient code, as well as debugging object microprograms. The system has been tried on both vertical and horizontal microinstruction designs and was found to produce very efficient code [BAB]. This system seems more efficient than SYSMW, but it does not provide the same interactive facilities that make SYSMW much easier to use without having had much previous microprogramming experience.

SUILVEN [SOM] is a high-level microprogramming language that generates code for a B1700 computer. In contains no machine independent features but the authors state that the programs may be altered so that SUILVEN may be run on another machine [SOM]. SYSMW's microprogram development system is totally machine independent and is therefore capable of producing microcode for any microprogrammable machine.

Monchaud et al. [MON] have developed a low cost microprogram development system and loader. The system uses an existing INTEL MCS 80 kit intended for use with the Intel 3000 series. The added 8255 I/O chips and the availability of pre-defined fields in the microinstruction are supplied at a cost of around $3000 [MON]. This does not include the cost of the Intel MCS 80 kit. The total cost of SYSMW is in the region of R500.

The SUMA [COR] microprogramming system is intended to fulfill the role of a high level language programming system that can be easily tailored to a specific target machine. The system requires a description of the target machine, followed by a specification of the algorithm which must be translated to execute on that machine. This system is similar to SYSMW but it is not menu driven, and the machine definitions are quoted as being 'time consuming' [COR]. In fact all machine independent, high level language systems require a description of the target machine, and in most cases this is a very tedious task. SYSMW, being a helpful, menu driven interactive system, makes the task of describing the target machine an easy one.

Skordalakis [SKO] proposes using an 'adaptive microassembly language' and a 'meta-microassembly language' to solve the problem of producing a more flexible microlanguage for bit-sliced microcomputers. For a particular processor, the programmer has to customise the adaptive microassembly language to its needs, using the meta-microassembly

language. The customised microassembly language can then be used to write the required microprograms. An example is shown for a horizontal type architecture where the microinstruction is divided into fields and subfields. The main design criterion here is that of lessening the burden of learning and using the system, but it is still not interactively menu driven [SKO].

The development aid programs suggested by Colard [COL] are designed specifically for use with the Am2900 series only. The system contains a general specification section while the microprograms are written in a symbolic language. A simulation package is also included, along with a small optimization program that checks for pairs of bits that always have complementary values or single bits that always have the same value. This implementation is however machine specific [COL].

REGTRAL, designed by Ballieu et al. [BAL], is a microprogramming language at the register transfer level. Its design objectives are minimal prerequisite hardware knowledge and readable and structured microprograms. The authors felt that they had to find a suitable language for horizontal microprogramming which should satisfy the following requirements: it must be easily readable, it must be extensible and adaptable to future needs, it must be useful for a great variety of applications, and it must allow full use of all the features provided at the hardware level. It was for these reasons that a high-level language was not chosen, but rather the class of register transfer languages, where there is a one to one correspondence between register transfer instructions and microwords. (It is interesting to note that the authors admit that high level languages are not suited for microcode specification, in the face of the fact that many high level language microprogramming systems have been developed and used. Surely only those high level languages that explicitly support concurrent constructs, can be suitable for this type of application.) The basic construct in a register transfer language is the micro-

operation, which can perform one logical function. Each microinstruction consists of a set of micro-operations and it is the microprogrammer's task to group the micro-operations together. The microprogrammer must have some knowledge of the functional structure of the computer and of the microword organisation. This system can only run on the PDP 11/60 computer [BAL2].

MMDS [MEI] is a microprocessor based microprogram development system, where special attention has been drawn to a user friendly interface design. The system, as described, depends heavily on the specific features of the target machine used, but it can be adapted for use with other systems. The microprogram production involves various tasks including, an editor for the input and update of the source programs, a compiler or assembler to produce the object code, a linker and loader, a debugger, and an overall test system [MEI]. User friendliness is also one of the main design criteria of SYSMW.

Mezzalama et al. [MEZ1] describe a hierarchical integrated system aimed at microcode development. The system is based on three different tools; a microprogram meta-assembler, a general purpose microcode simulator, and a microprogram compiler to convert a high level language into microinstructions [MEZ1]. This system, like SYSMW, is machine independent and can be used to develop microcode for a variety of machines.

AMDASM, first produced by Advanced Micro Devices, is the most widely used commercial micro-assembler system. MetaStep [DAT] speeds up the AMDASM assembler and allows the user to write with case structures instead of with low level symbols and bits. Metastep is considered to be the fastest and most efficient assembler to date, at a cost of around $23 000 [DAT]. AMDASM is supplied with AMD's SYS 29, which is discussed later in this section.

Dimond et al. suggest a flexible development system for microprogrammable microproccessors. A flexible prototyping unit is constructed using a host minicomputer for use with a range of microprogrammed processors. The host computer is interfaced to the arithmetic logic unit (ALU) and to the microprogram control unit (MCU) separately. 'BASIC', which is interpreted, is used to control the whole system which therefore runs very slowly. The microprogram memory is just an area in main memory and the system is able to execute a single microinstruction at a time [DIM]. SYSMW uses the popular IBM PC as its host computer.

The 'AMD 2900 learning and evaluation kit' has various limitations as a laboratory microprogram instruction kit. Inputs are entered via toggle switches and the outputs are shown on a row of LEDs. The system only caters for 16 words of 32 bit microprogram memory and the ALU is only 4 bits wide. Davies et al. [DAV] have produced a system that improves the user interface with the AMD 2900 kit by providing it with a simple software monitor. SYSMW is far more useful in the laboratory environment, in that it not only allows the user to experiment with microprogrammed control, but it also provides a microprogrammable emulator for higher level languages.

Morgan et al. [MOR], at the University of York, have recently developed an educational bit-slice microprogrammable tutor. The processor is based on the Am2900 series and consists of an arithmetic logic unit and a microprogram control unit. A host microcomputer based on an 8085 CPU emulates a very slow microprogram memory. The inputs to the ALU are controlled by a set of switches and the output is displayed on a row of LEDs [MOR]. The system can therefore only be used to illustrate the concepts of microprogramming and, unlike SYSMW, cannot emulate any real life macroinstructions.

Sridhar et al. [SRI] suggest an automatic microcode generator for high

level language machines. The microcode generator takes as input the description of language to be implemented, in the form of the BNF productions for that language. A microcode compiler then generates the microcode in terms of a microassembly language. 'With modifications, this can be used to generate the actual microcode which defines the ROM bit fields' [SRI]. The output is therefore just a brief description of the various hardware functions needed to accomplish a given task in a certain language, and does not constitute the final microprogram.

3.1.2 : AMD's SYS 29.

Advanced Micro Devices' SYS 29 is a complete development system that encompasses all the tools needed, from microcode definition, assembly, checking of the hardware, formatting of the microcode, through to programming PROMs [ADV]. The system includes a mainframe, VDU, dual flexible disk drives, and a comprehensive software package. The basic configuration includes 2K of 64 bit words writeable control store and a CCU microprogram controller card. The writeable control store can operate at clock speeds of up to 18,Ø MHz [ADV].

AMD's SYS 29, while on loan to Rhodes University, was used by the author.

The host processor contains the Am9Ø8Ø CPU which has the same fixed instruction set as the Intel 8Ø8Ø microprocessor. There is 32K of RAM available to the user for the storage of various routines. It is on this hardware that AMDASM, the resident microassembler, is run. AMDASM allows microprograms to be written, assembled and loaded into the control store.

Without the optional CCU card, the system only provides the control store of a microprogrammable processor. Even with the extra CCU card,

the user still has to develop his own bit-slice ALU, on a prototyping card, before the system can operate as a microprogrammable machine. This makes it very difficult for the user to test his ALU hardware. Unless he develops a simulator, and is then able to use this to ensure that any 'bugs' do not lie in the microcode itself, he is unable to validate his hardware design.

The AMDASM system allows the target machine definition to be entered into a definition file. The microinstructions are then written using these definitions, with the system allowing one microinstruction per line. The assembler takes this file of microinstruction mnemonics and produces the binary file for loading into the control store. This process of microprogram development is very similar to the approach taken in SYSMW. SYSMW however allows the definition of the target machine and the microprogram, to be accomplished in an interactive, menu-driven type environment.

3.1.3 : Simulators.

Microprogramming is difficult and error-prone and particular difficulties are encountered in the testing and debugging of the microcode. Some of these problems may be overcome by the use of a simulator before introduction to the target machine. A simulator based microcode development system may also become a valuable teaching and training aid. The greatest potential benefit of microcode simulation is in the scope it offers by removing the constraints of the real hardware environment for the provision of testing and debugging aids. Charlton et al. [CHA] discuss a microcode development system to assist in the production of microcode for a range of 16-bit processors made from Am2900 series bit-slice components. The system places emphasis on interactive microprogram testing and monitoring, and because is it simulator-based, it is inevitably machine-dependent. Features are

included in the system to assist the programmer, designer or student to manage the demanding intracacies of microcoding [CHA].

Myers et al. [MYE2] suggest a simulator that interprets the microprogram, providing exactly the same state changes and effects (except for elapsed time) that would occur if the microprogram existed in its actual environment. They list the following motivations for using a simulator for microprogram testing and debugging: it allows parallel microprogram and hardware development, it allows easier debugging due to special facilities, it allows special error checks (one does not normally have logic in the hardware to detect microprogramming errors), it allows several people to use the simulator in parallel to develop different parts of the microprogram, it is easy to change the microprogram as you go, and one can implement the calculation of performance statistics. The simulator can also be used as a standard against which the hardware may be checked once completed [MYE2].

A strategy for simulating bit-slice based microprogrammable systems gave rise to a system specifically orientated to the development of highly horizontal microprogrammed machines [MEZ2]. The proposed simulator has a 'certain degree of generality' although the specific example sited is based on the Am2900 series. The description of the machine structures is separate from the simulation procedures. The system has powerful interaction capabilities between the user and the simulated microprogram [MEZ2].

Because the microprogram control unit in SYSMW's Triple-M machine is simulated, the user is able to benefit from the advantages that this provides. He is however not sheltered from the 'real world', in that he still has to control the Am2901 based ALU hardware. Control of both sections is necessary for complete control of the system.

3.2 : Survey and Recent Developments in Bit-Slice components.

The following paragraphs on various bit-slice chips have been included for completeness, and give the reader a brief glimpse of other available bit-slice technologies.

The INTEL 3000 series was one of the first bit-slice families to become commercially available [RAU]. The family consists basically of the 3001 sequencer and the 3002 processing unit, which is two bits wide but may be cascaded to any desirable width.

Monolithic Memories' MMI 5700/6700 family consists of a microprogram controller 57110/67110, and a processing element called microcontroller-5701/6701 [RAU]. The processor elements are four bits wide and special features include a full internal carry look ahead and an auxiliary Q register for multiply and divide implementations.

The Motorola 10800 processor family includes a MC10801 sequencer and a MC10800 four bit ALU slice [RAU] [MCG]. The M10800 family is implemented in ECL technology and is therefore capable of very high speeds.

Texas Instruments SN 74S481 is a four bit microprocessor slice developed using Schottky bipolar technology [RAU] [MCG]. It is one of the most powerful processing elements on the market, due to special high-level functions, multiport architecture and multioperation design. The SN 74S482 is a microprogram sequencer that is capable of being used with many of the popular processing elements on the market.

The Fairchild F100220 family was announced in 1980 and is implemented using ECL bipolar technology [MCG]. The ALU slices and supporting chips are all eight bits wide.

Advanced Micro Devices' Am2900 family is used as a basis for SYSMW's Triple-M microprogrammable machine. The newer Am2903 is a very fast, four bit microprocessor slice that performs all the functions of the Am2901, together with additional arithmetic functions [MCG]. These functions allow multiplication and division to take place more efficiently. Another key feature of the Am2903 is that the number internal working registers may be expanded using the Am29705 external register stack.

More recently, Texas Instruments have developed the TMS 7000 series of eight bit single chip microcomputers [TEX]. The TMS 7000 is hailed as the world's first microprogrammable microcomputer that allows the standard instruction set to be changed to suit the needs of a particular application. The architecture has been optimised for fast real time applications and is very powerful.

Hewlett-Packard's HP 3000 Series 37 is another recent example of a microcoded processing unit [AME]. The microprocessor is very powerful and flexible, and this flexibility causes the microcode to be complex and hard to understand. A translator is used to create a more readable microprogram.

4 : DISCUSSION AND CONCLUSIONS.

4.1 : Discussion.

During the last thirty years microprogramming has evolved through several stages. It started as a concept for implementing the control sections of processors, then an economical method for manufacturer implementation of machine architecture, and more recently as a tool for user applications for specific problems [RAU]. Advances in hardware technology have made user microprogrammable machines a reality, and support tools have simplified the tasks of the microprogrammer.

Hardware support tools for microprogram development have come a long way but advances in software and applications aspects of microprogramming are still in their infancy [RAU]. The SYSMW development system attempts to satisfy some of the needs for further research and development in this area.

The SYSMW microprogram development system is not intended for use in areas where large amounts of microcode have to be produced. This task becomes very tedious without the aids that a high-level microprogramming language system can provide. One of the main drawbacks of using a high-level compiler is the possible inefficiency of the resultant microcode. Efficiency depends largely on the complexity of the translation algorithms, how suited the high level language is to microprogramming constructs, and how well the user has defined the target machine. The complexity of the translation programs will depend on the specific package used. In most development systems the definition of the target machine tends to be a very tedious task, and the user could therefore be tempted to use a simple, less efficient definition. The efficiency of the code produced by the SYSMW development system is user dependent as the definitions are made at a very low level.

The choice of a suitable language to be used as a medium to develop the microcode is paramount. Microprograms require an increasingly large effort for their design, due to the requirement of maximum utilisation of concurrently usable resources. (Microprograms become larger as more and more software functions are vertically migrated.) A suitable language should therefore allow the representation of all possible parallel operations during the design process. Kerner [KER] suggests that there are some similarities between the demands of microinterpreters and parallel computers and therefore looks at recent solutions in this area for answers about the design of microprograms. He suggests that the new 'data flow principle' appears to be suitably adaptable to microprogramming. The underlying idea here is the use of functional languages for the design of microprograms and microcode interpreters. It seems certain that languages that do not explicitly support concurrent constructs cannot be used effectively in the definition of microcode segments.

One of the future trends in microprogramming could be the increased use of microprogrammable machines that have been microprogrammed to execute high level language instructions directly. The two main areas of interest to date have been the implementation of 'FORTH' machines and 'LISP' machines. The LISP programming language has the same general property as machine code in that data and programs are indistinguishable [PUT]. In most cases, the LISP instructions are translated into some form of intermediate language, which is then interpreted by the microprogram [PUT] [DUE] [GRI].

The topic of microcode optimisation has not been discussed in this report as it falls outside the scope of the project. Various methods of optimisation have been suggested, mostly peephole optimisation on the complete microprogram. Simple tests can be made for single bits that always have the same value, or for pairs of bits that always have

opposite values [COL].

4.2 : Conclusions.

SYSMW is a complete microprogram development system that allows the user to experiment with most aspects of microprogrammed control. It is ideally suited for use in an academic environment, to provide the student with the necessary practical experience - vital to the understanding of microprogramming techniques. The microprogram development aids may be used to develop the microcode for other microprogrammable machines, while the Triple-M microprogrammable machine may be used to experiment with new forms of machine architectures. Research into viable intermediate language designs, for fourth or fifth generation languages is also possible. More specifically, it would be nice to implement a set of intermediate codes to interpret a LISP (or LISP type language) based system. It would also be possible to build up the primatives for a language like FORTH, so that the microprogram could emulate the FORTH primatives directly.

Raucher et al. recognise that microprogramming language techniques, such as intermediate language design, interpretation, and optimisation need to be applied and extended, specifically in the field of bit-slice processors [RAU]. This leaves the field of possible research projects, using a development system like SYSMW, wide open.

SYSMW has a distinct advantage over other systems described in this report, in that it is very easy to get a new architecture design up and running very quickly, using a subset of the final design. The testing of a new system is therefore a lot easier, and it is possible to develop the system in a logical, extensible fashion, until the final goal is reached. There is a similarity between the stepwise refinement

techniques of software development and the techniques fostered by SYSMW.

To support the claim that SYSMW has achieved its design goals, it seems appropriate to quote from a few of the reports of the Honours Class students, involved in the testing of the system. These quotes are listed below:

'The system was remarkably easy to use and understand.'
'The simplicity of the system is commendable.'
'It is both a powerful and effective tool, and is a great aid to understanding microprogramming.'
'The system has much potential in helping the user develop microcode.'
'One is conscious of the fact that the author has attempted to make the system as user friendly as possible and I feel he has succeeded in this.'
'I found the microprogram development system extremely helpful and instructive from the point of view of someone who has only just started microprogramming.'
'The development system is indeed a useful tool for the specification of microcodes.'

The future of SYSMW, at least at Rhodes University, seems secure. It is to be used as the basis for the practical component of a post graduate course in microprogramming. Interest has also been shown in using the Triple-M machine to run a small assembly type language, as part of an undergraduate course in compiler design and code generation. Other areas for research (such as investigating LISP intermediate-codes) have been opened up by the SYSMW project, as there has previously been no similar resource available. SYSMW is particularly attractive in that it runs on the very popular IBM PC and can be produced for about R500.

PART 2

1 : DESIGN DETAILS.

1.1 : Introduction.

The overall design specifications of the SYSMW microprogram development system are covered in the sections that follow. It is assumed that the reader has seen Part 1 and that he is familiar with the terms and concepts involved in the implementation.

The whole system can be divided into two main sections. The first of these deals with the development of the microcode. The programs involved here are Micro1 and Micro2. Together they form a flexible microprogram development system which can be used to write

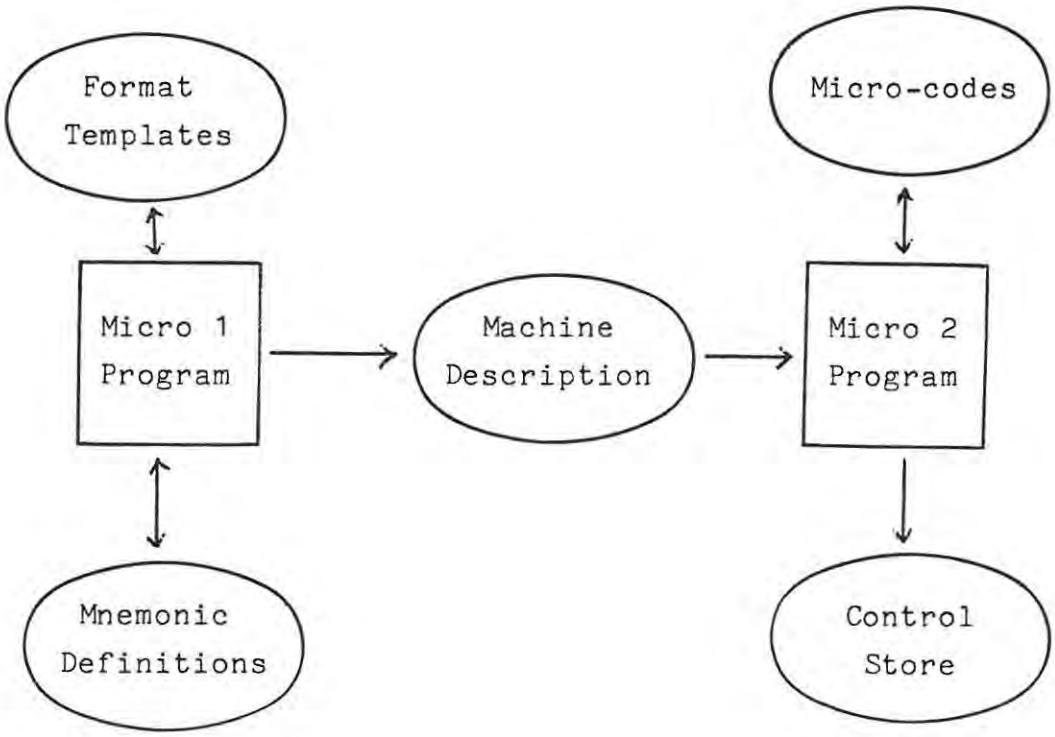


Figure 1.1
THE MICROCODE DEVELOPMENT SYSTEM

microprograms for any microprogrammable machine. The machine-independence of the microprogram development system can be attributed mainly to the low level at which definitions are made. Figure 1.1 shows the basic layout of this system.

The second section involves the microprogrammable machine itself. The 'machine' is in fact part hardware and part software simulated. The hardware consists of an arithmetic and logic unit (ALU) built using components from AMD's 2900 bit-slice family. The software emulates the microprogram control unit (MCU) and also interfaces to the control store (CS). The ALU, MCU and CS together form the microprogrammable machine. A software simulator that produces the effect of the hardware and therefore allows microcode testing to be carried out independently of the hardware, is included for debugging purposes. Figure 1.2 gives a general outline of the microprogrammable machine.

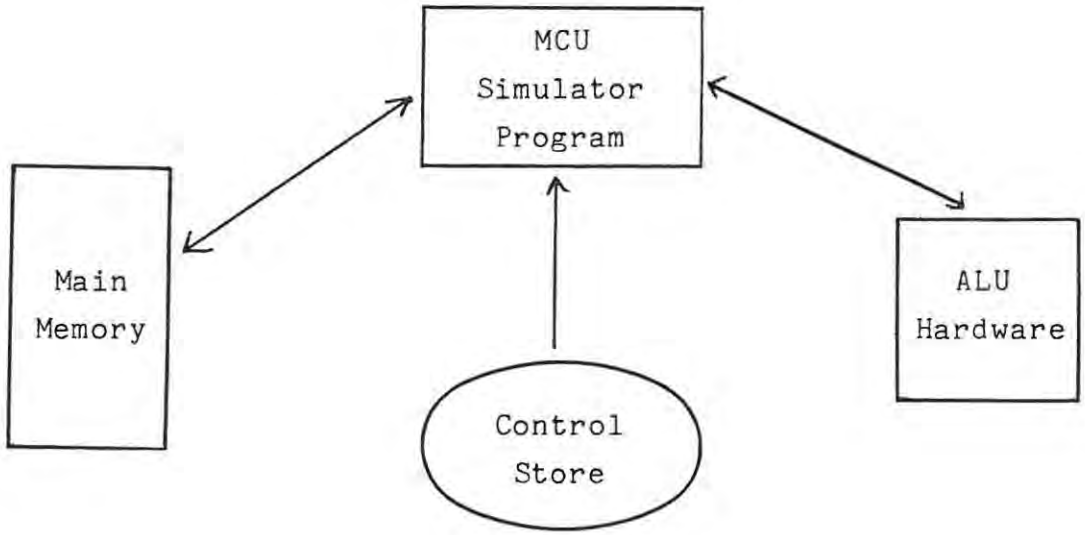


Figure 1.2
THE MICROPROGRAMMABLE MACHINE

1.2 : The Target Machine definition Program.

Micro1, the first of the microcode development programs, enables the user to describe the target machine, ie. the machine on which the microcode is going to run. This is done by setting up two files, called the format file and the definition file. The only other vital information required is the bit width of the microinstruction.

A third file is generated by the program and contains information from both of the input files. This table file, containing the basic machine definition, is used as input to the second microcode development program, Micro2. The basic file layout of Micro1 is shown in figure 1.3.

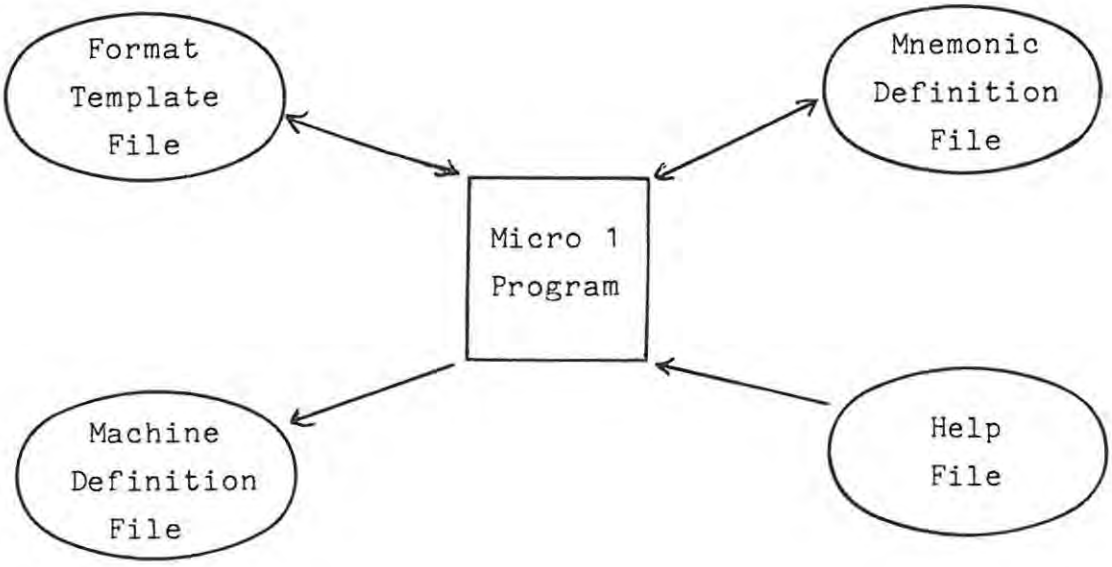


Figure 1.3
THE MICRO1 PROGRAM FILES

The Micro1 program is designed around the following basic algorithm:

```
Begin
   Assign all the necessary files
   Check to see if the helpfile is to be included
   While it is not the end of the session do the following
      Print out the main menu
      Then either  set up a new file
                Or  update an existing file
                Or  print out one of the files
                Or  set up the table file
                Or  ask for help
                Or  terminate the session
   Close all the assigned files
End.
```

A few preliminary functions have to be performed before the core of the program code may be executed. The format, definition and table files have to be linked to their real file names. The helpfile facility may then be included at the request of the user.

The main menu options, printed out along the top of the screen, help the user to select the desired operation. When using the system for the first time, new format and definition files would have to be created. The user is led through the various routines with the aid of prompts, running under a menu driven system. The information in either of these files can then be updated if necessary. These updates are carried out on one record at a time (the record structures are discussed later in this section) and again the user is led through the routines with the aid of menus and prompts.

'TURBO' Pascal [TUR], the language in which all the SYSMW software is written, differs from standard Pascal in the way in which it handles

its files. In standard Pascal, the files may be opened for input only or output only. In TURBO however, the files may be opened for both input and output at the same time, and information in a record may therefore be read, updated and re-written very easily. TURBO also caters for the random access of records in the files using a 'seek' function, and records are therefore easily found, read, updated and re-written to the files.

What follows is a brief outline of the composition of the three files used during this phase of the microprogram development.

The format file consists of records that have the following structure:

(1) A format number, which is a unique number for each different format.

(2) A description, which is a short string that helps the user to identify the specific format.

(3) The actual format, which contains the information regarding active bits, ie where the bit pattern using this format is to be placed in the microinstruction. It therefore defines where the various micro-orders are to be placed within the microinstruction.

(4) A delete field, which is a Boolean variable that records whether the record has been deleted or not.

(5) An integer field, which holds the length of the active fields for the format.

The fields of the formats entered by, and visible to the user, are shown below:

```
Format number: 6
Description:   2901-alu-function
Format:        42x 3a 11x
               (an x corresponds to a don't care field)
```

This implies that mnemonic definitions using this format, would have 3 bit wide binary values that fit into a 56 bit wide microinstruction, starting at position 43.

The definition file consists of records that have the following composition:

(1) A format number, which is an integer which points to the format description that this mnemonic definition uses. This field forms the connection between the records in the format file and the records in the definition file.

(2) A mnemonic, which is a string that holds the mnemonic name. These names would normally be chosen such that they clearly reflect the function that is to be performed. This enables the microcode to be understood more easily.

(3) A value field, which holds the binary value associated with the mnemonic. This value field will be inserted in the microinstruction at the micro-order position given by this mnemonic's associated format.

(4) A delete field, which is a Boolean variable that records whether that record has been deleted or not.

These mnemonic definitions are generated by the user in the form given below:

```
Mnemonic:   AND
Value:      110
Format no.: 6
```

This example mnemonic 'AND' uses format number 6 as a template and has the binary value '110'. Other mnemonics could also use the same format template.

The structure of the table file records is given below:

(1) A mnemonic from the definition file.

(2) Its associated binary value from the definition file.

(3) The corresponding format from the format file. This format is obtained by using the format number associated with the mnemonic and then searching the format file to find the appropriate format template as explained below.

The format and definition files may be listed to either the screen or a printer, enabling the user to verify their contents. Once the user is satisfied that the information contained in these files is correct, the procedure to generate the table file may be invoked. This routine works through the definition file one record at a time collecting the necessary information for a table file record. The 'mnemonic' and the 'value' are obtained from the present definition file record, while the 'format' has to be looked up in the format file. The 'format number' from the definition file record provides the necessary link to enable the correct format to be inserted into the table file record. The records are written to the table file one at a time, as they are set up. Checks are made to see that the 'active' part of the format and the length of the 'value' field are the same. An error is also reported if the corresponding entry in the format file is not found.

For the example of the mnemonic 'AND', the table file entry would read as follows:

Mnemonic: AND
Value: 11Ø
Format: 42x 3a 11x

Whenever the mnemonic 'AND' is encountered in a microinstruction definition, the value '11Ø' will be inserted in the microcode starting at position 43.

The first entry in the table file contains the width of the microinstruction, completing the information necessary to describe the target machine.

When input is required from the user during the execution of Micro1, an option exists for invoking the help facility. The help message, read in from a help file, is listed on the screen. A help facility implemented in this way is particularly useful when there are numerous help messages as it makes the program code more readable. Messages in the helpfile are arranged in the order in which they are most likely to be requested during a typical session, to keep search times to a minimum. The help messages, when listed to the screen, overwrite the characters that are in that section of the screen. When the message is erased, the previous information in that section appears to have been lost. This problem is solved by copying the information to another section of the 'screen' memory, corrupting the screen with the help message and then copying the old information back to the active screen. (The implementation of this idea is discussed later in more detail.)

Once the target machine has been described using Micro1, the microcode for this machine can be developed using the Micro2 program. The table file is used to convey the target machine information to the Micro2 program.

A listing of the Pascal code for the Micro1 program is given in appendix A.1 and A.2.

1.3 : The Microcode definition Program.

Micro2 is the second of the programs in the machine-independent microprogram development system. It allows the user to write the microcode more easily, using the mnemonics that were defined during the execution of the Micro1 program.

The table file that was set up during Micro1, is used as input to this program, along with the user generated file of microcodes. The program produces several output files, these include a label file, a decode file, a control store file and a hexadecimal instruction file. Refer to figure 1.4 for a basic layout of the files used.

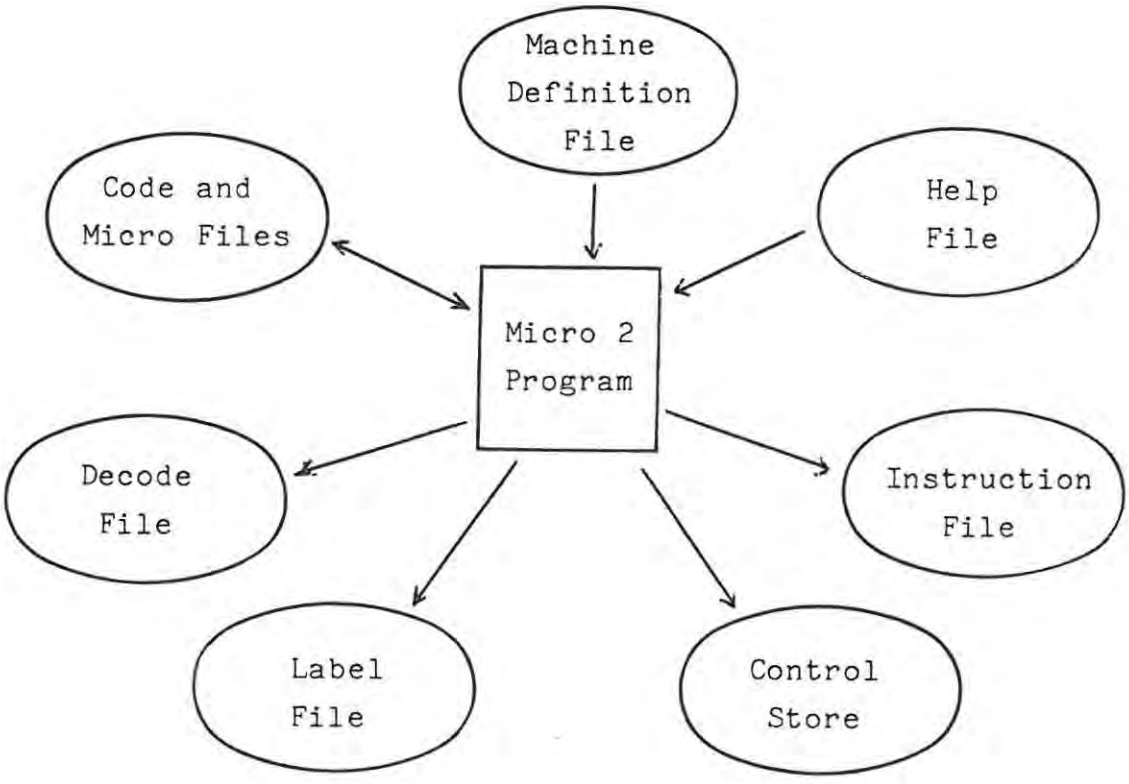


Figure 1.4
THE MICRO2 PROGRAM FILES

The algorithm on which the Micro2 program is based is as follows:

```
Begin
   Enter the name of the table file.
   Then include the helpfile if requested.
   While it is not the end of the session do
      Print out the main menu
      Then either  assign the workfile
                Or  edit the workfile
                Or  list the files
                Or  assemble the microcodes
                Or  ask for help
                Or  terminate the session
   Close all the assigned files
End.
```

Before the main part of the program can proceed, a few precursory functions have to be performed. The name of the table file, from the Micro1 program, has to be entered and all files have to be linked to their real names. The inclusion of the helpfile facility may be requested.

The user is able to select one of the options given in the main menu. The first task would be to assign the work file name . The work file is the name given collectively to the file of user entered microinstructions and the other files generated by this program. On using the system for the first time one would have to set up the file of microinstructions. This file is created using the user-friendly, menu driven editor resident in Micro2. The microcodes entered in this way are stored in memory in the form of a dynamically linked list. Two physical files are needed to store the information on disk and these are called the code file and the micro file. The code file consists of records that contain the macro-instruction mnemonic, the number of

microinstructions for this macro-instruction and various other fields. The macro-instruction would be a command in the language that the microprogram is intended to emulate.

The micro file holds all the microcodes, stored in such a way that if the first macro-instruction has five microcodes then the first five records in the code file hold these microcodes. If the second macro-instruction has three microcodes then the next three records in the micro file hold these microcodes and so on. Figure 1.5 illustrates this structure. These two files are collectively referred to as the macro-instruction definition file. The updating facilities allow the contents of this file to be inspected and altered if required. The 'random access' facilities of TURBO Pascal were used to simplify this operation as is described in the section on the Micro1 program.

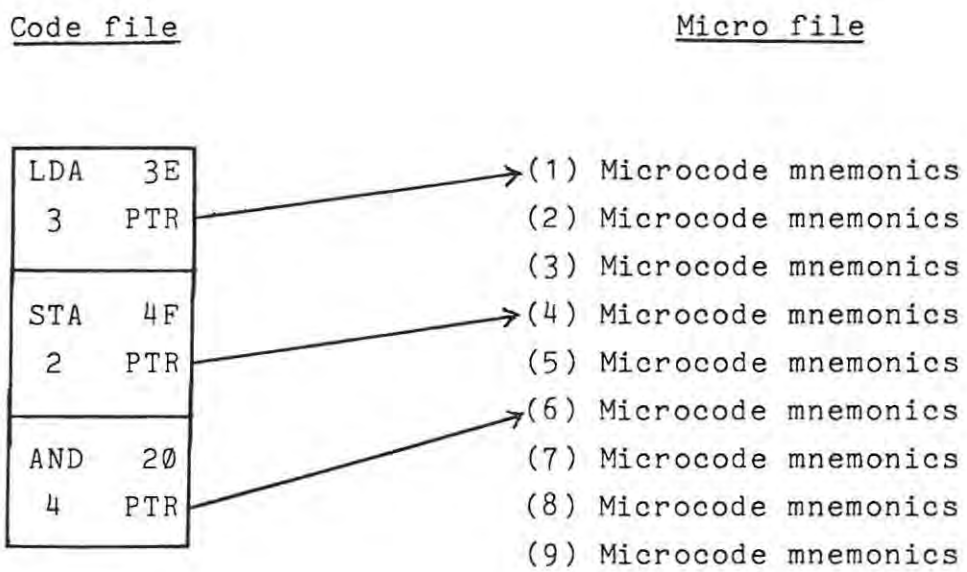


Figure 1.5
LAYOUT OF THE CODE AND MICRO FILES

Once the microcodes have been entered correctly they can be assembled into binary micro-instructions for loading into control store. The assembly is performed by a two pass assembler. It accepts input from the table file and from the macro-instruction definition file. A jump table file is used by the micro-assembler to patch forward references during a second pass. The decode file, also generated during the assembly, acts as the look up table when the microprogram is running, as it allows the opcode to be exchanged for an address in the microprogram where the microcode sequence for that macro-instruction is to be found. The hexadecimal file contains the microcodes in hexadecimal form, as this is sometimes easier for debugging purposes. The assembler reports errors if duplicate labels have been defined or if there is a jump to an undefined label within the microprogram.

What follows is a brief outline of the structure of the 6 files used during this phase of the microprogram development.

The table file has already been discussed in the section on the Micro1 program.

The macro-instruction definition file consists of records that have the following composition:

(1) A string field holding the name or mnemonic of the macro-instruction or the micro-routine. A macro-instruction is a command in the language which the micro-code is to emulate while a micro-routine would be an associated function needed in that language eg. a startup or boot routine or a fetch cycle in an assembly language implementation.

(2) The actual hexadecimal opcode associated with this macro-instruction mnemonic. A micro-routine has no associated mnemonic.

(3) A delete field, which is a Boolean flag that records whether the record has been deleted or not.

(4) An integer field that stores the number of microinstructions for this macro-instruction or micro-routine.

(5) A pointer to the first microinstruction in the sequence of microinstructions for this macro-instruction.

Each of the microinstructions are held in records that have the following structure:

(a) A microcode field that contains the micro-order mnemonics for this microinstruction. These mnemonics were defined by the user during the execution of the Micro1 program.

(b) A position field that holds the random access position of the micro-order mnemonics in the table file.

(c) A pointer that points to the next microinstruction for this macro-instruction or micro-routine.

The information in these records is presented to the user in the following format:

```
Macro-instruction:   LDA
Hexadecimal opcode:  3E
Micro-codes:
Areg #0000 & Breg #0000 & RAMA & ZB & ADD & CO=1 & I-U & YB-AR
DR-DB & DZ & OR & Breg #0001 & RAMF & I-M & CY
NOP & JUMPADDR # fetch & NOINCR & UNCOND
```

The system assumes that the micro-order mnemonics that form one microinstruction are all on the same line.

The structure of the records in the decode file is shown below:

(1) A mnemonic field that contains the macro-instruction mnemonic.

(2) An opcode field that stores the opcode associated with the mnemonic.

(3) A microaddress which is an integer that points to the position in the control store where the microcode for that instruction

starts.

When listed, the information in these records appears as follows:

Mnemonic: LDA Opcode: 3E Address: 19

The label file is a temporary file used by the two pass micro-assembler and each record contains the name of the label and its corresponding address.

The instruction file, which contains the hexadecimal equivalents of the binary microinstructions, is simply a file of text.

The binary file of microinstructions is a file of arrays, each consisting of 31 bytes. The first byte contains the width of the microinstruction while the remaining 30 bytes forms a string of 240 bits which holds the microinstruction.

The macro-instruction definition file, the hexadecimal file of microinstructions, the binary control store file, and the decode file, may all be listed to either the screen or the printer.

During the execution of Micro2, when input is required from the user, an option is available to invoke the help facility. The help facility has been implemented in the same way as described for the Micro1 program.

A listing of the Pascal code for the Micro2 program is given in appendices B.1 through B.3.

1.4 : The Development System in Use.

The Micro1 and Micro2 programs together form a machine-independent microprogram development system that can be used to develop microcode for any microprogrammable machine. This applies to both vertical and horizontal microinstruction designs. The extent of the flexibility of the system is demonstrated by the fact that it could even be used to assemble fixed-instruction set mnemonics for a monolithic machine.

As has been shown, Micro1 is used to describe the target machine while Micro2 is used when developing the actual microcodes. These two programs were used as the basis of a practical assignment, for the Computer Science honours class at Rhodes University.

The assignment involved the writing of a few microinstructions for a simplified computer processor. The students were told to assume that any requirements for branching within the microprogram may be neglected and that all timing requirements for preventing race conditions had been satisfied. The assignment was to be accomplished using Micro1 and Micro2 and involved three stages. Firstly the schematic diagram of the hypothetical processor had to be studied and suitable formats for all necessary micro-orders had to be designed. Secondly students were to use the micro-order tables provided, to define mnemonics for the codes that they would require in their micro-order files. The tables were of the type found in the standard data sheets which accompany bit-slice chips, and so made the exercise more realistic. Thirdly they were to use their mnemonic definitions to write microprogram segments to perform an 'instruction fetch cycle', a 'load immediate value into the accumulator', and a 'load the value found at a given address into the accumulator'.

On the whole the students managed to familiarise themselves with the microprogram development system easily, and most of them were able to

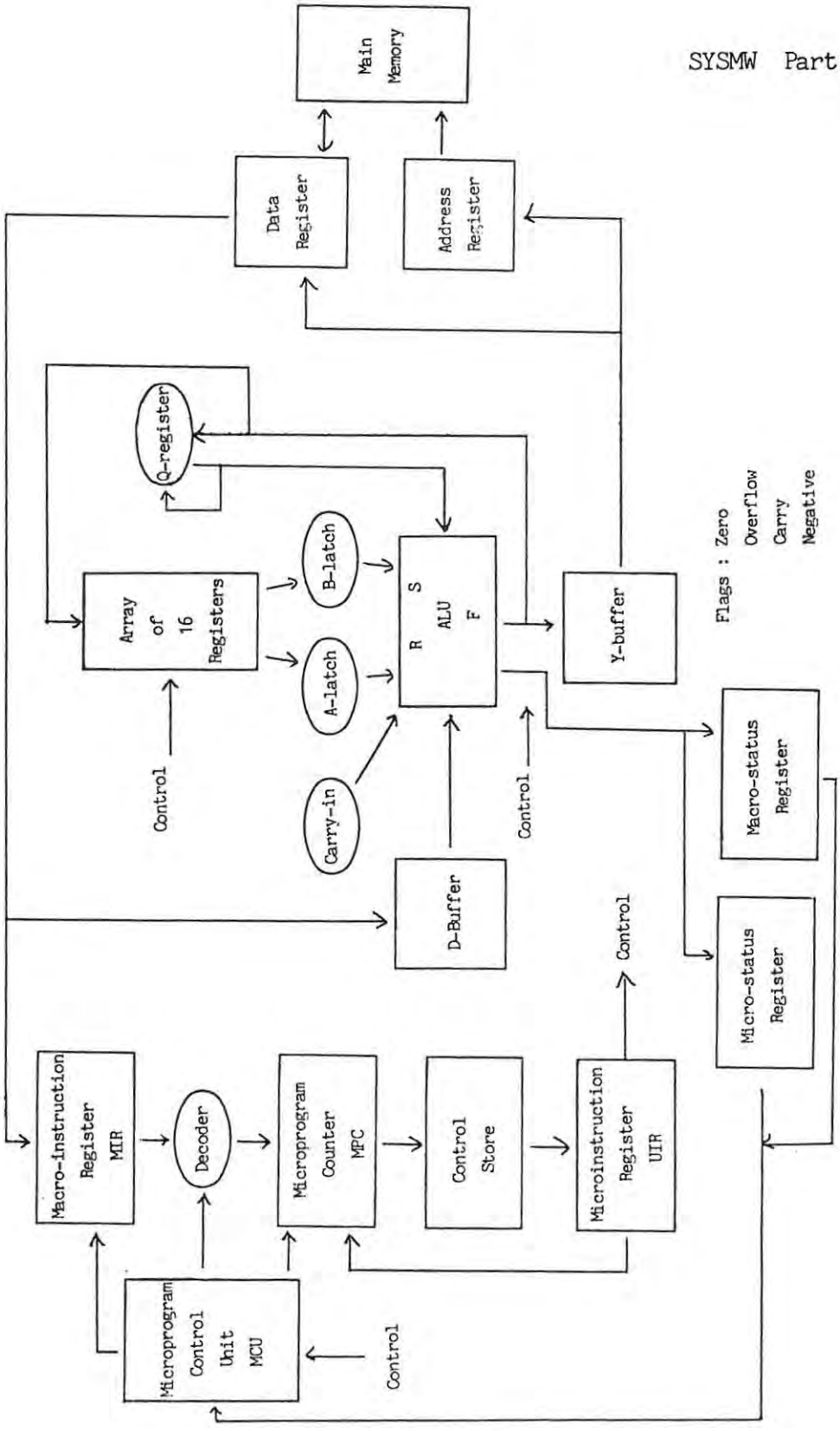


Figure 1.6

SCHEMATIC DIAGRAM OF THE PROCESSOR CIRCUIT

complete the prescribed tasks.

Figure 1.6 is a copy of the schematic diagram of the 2901 based processor circuit that was used in the exercise.

The students were then asked to write a 'discerning article on a user's view of the microprogram development system'. A summary of the main points raised, in the seventeen articles received, is given below. In the case of suggested system changes, brief reasons are given as to why these were implemented or not.

Points which received favorable comment:

(1) The double check at the end of an update session to see if the user was sure that he wished to delete the records was a good safeguard.
(2) The help files were very useful.
(3) The help messages appeared on the same screen from which they were called, making the interpretation of the help messages a lot easier.
(4) The option of being able to list files to either the screen or the printer and the check to see if the printer was ready, was good.
(5) The prompt menus were very useful and quite self explanatory.
(6) The good menu layout and simple hierarchy of the system made it easy to use.
(7) The error messages were very useful and it was good that some checks were made before actual assembly.
(8) This was an excellent way to be introduced to actual microprogram writing.
(9) It was found that the development system was extremely useful for someone who has only just started microprogramming.
(10) It was a good tool for the specification of micro-codes.
(11) The simplicity of the system was commended.

Criticisms and general suggestions that were worth implementing.

(1) The programs aborted if the input files were not found. This has been changed to report a non-existing file. If no file name is entered, the program now terminates normally.

(2) The help message sometimes erased part of what was already on the screen. A system of screen copying had to be implemented to alleviate this problem. A more detailed explanation of this feature follows at the end of this section.

(3) One should have been able to insert and delete microinstruction lines during the update routines. This has been implemented.

(4) If a duplicate format was entered, nothing was said as to what action had been taken. The system now reports the error and ignores the duplicate.

(5) The system needed a user manual to elaborate on the system functions and the sometimes cryptic help messages. A user's guide appears as a section in this report.

(6) There should have been a system start screen that would give the user some introductory information. A page giving the name of the program and the details of the author appears on the screen at the start of each program. This is followed by a brief explanation of the program including information on the files needed for that program.

(7) One should have been able to stop assembly by pressing a key. Holding the space bar down now halts assembly. The user is then asked if he wishes to abort or continue the assembly.

Criticisms and general suggestions that were not worth implementing.

(1) Only one line of micro-order mnemonics per microinstruction was allowed. It was felt that this was not a serious constraint. Some form of line extension symbol would have made the micro-codes less readable.

(2) Files could only have records inserted at the end. This has no influence on the smooth running of the system and was only

suggested for aesthetic reasons.

(3) The system needed to be more user friendly. There is a balance between user friendliness for the new user and the tedium of having too much text on the screen for the more experienced user. The user's guide, not available at the time, should also assist in complementing the sometimes cryptic help messages.

(4) One could not change the microinstruction width from within the programs. It would be dangerous to allow this kind of change as the width of the microinstruction is a fundamental quantity and should be decided upon before embarking on microcode development.

(5) There were no checks to see if the binary numbers entered amongst the micro-order mnemonics were the same length as their associated format fields defined earlier. The system would be significantly slowed down if such a check were implemented, as a further file would have to be searched and records checked.

(6) There should be no distinction between upper and lower case. This is a contentious point as some feel there should be a distinction and others do not. It was decided to leave the decision to the user who could use either upper or lower case for the mnemonic names.

(7) The questions needed to be more user friendly. Again there is a play off between user friendliness and too much text.

(8) More help information in the help messages was necessary. The user's guide should provide the necessary extra information.

(9) Another kind of separator between the micro-order mnemonics should have been used instead of the '&'. This is a personal preference and it was therefore decided to leave the '&'.

This broad cross section of suggestions has been included in this report to give a general idea of the usability of the system.

The screen copying system, that allows the help messages to be listed on the same screen from which they were called, deserves further

explanation. The 'IBM PC' reserves space in its memory for four screens. Normally only screen Ø is used as this is initialised as the active screen on which all operations are performed. Screen Ø starts at address $B8ØØØ and screen 1 starts at address $B9ØØØ. By defining two arrays on top of these reserved areas in memory, using the 'absolute' command in TURBO Pascal [TUR], one can gain access to them. Before the help message is printed out, screen Ø is copied across to screen 1. Screen Ø can then be partially overwritten by the help message. When the space bar is pressed by the user to clear the help message, the information in screen 1 is copied back to screen Ø and the original data is therefore returned to the active screen. The cursor also returns to the point from which the help message was invoked.

Having developed the necessary microcodes, they can be verified through interpretation on a microprogrammable machine.

1.5 : The Microprogrammable Machine.

The following sections form a discussion of a microprogrammable machine called Triple-M (Mike's Microprogram Machine) built around the 'IBM' Personal Computer.

In general a microprogrammable CPU consists of three main parts, namely an arithmetic and logic unit (ALU), a microprogram control unit (MCU) and a control store (CS). This general layout has been discussed in Part 1.

The Triple-M's CPU consists of the usual ALU hardware but is unique in that the MCU and CS are simulated by software running on an 'IBM PC'. One of the main features of this particular configuration is that the control store memory and the normal user memory are both in the same

physical memory space. The ALU hardware and MCU software communicate with each other via three Intel 8255 Programmable Peripheral Interfaces (PPI's). These PPI's each have 24 programmable Input/Output lines giving a total of 72 communication links.

The system is implemented with a single pipeline register called the Micro-Instruction Register (UIR). The microinstructions are loaded from the control store into the UIR and are then executed. The control signals for the MCU are therefore readily available while the ALU control signals have to be sent via the 8255 interface. The address in the control store of the next microinstruction to be executed is held in the Micro-Program Counter (MPC). Except in the case of a branch within the microprogram, the MPC is usually incremented to enable the next microinstruction to be fetched. It is possible to load the UIR form the Micro-Latch instead of from the control store. This Micro-Latch is used in the implementation of 'inline' microcode which is discussed in more detail in section 2.2. Here the Micro-Latch value is loaded from the main memory via the Data Register.

Figure 1.7 is a schematic diagram of the control paths of the Triple-M machine.

The data paths involve access to the main memory and access, via the interface system, to the ALU. An Address Register (AR) and Data Register (DR) have been implemented as buffers for the main memory and they can be seen as acting as the address and data busses of a conventional machine. If the value in the Data Register is an instruction, it is loaded into the Macro-Instruction Register (MIR). The system then decodes this instruction (using the information in the decode file from the Micro2 program) to calculate an address in the control store where the microcode sequence, to emulate this instruction, begins. The address is loaded into the MPC and the corresponding microinstruction is read into the pipeline register, UIR.

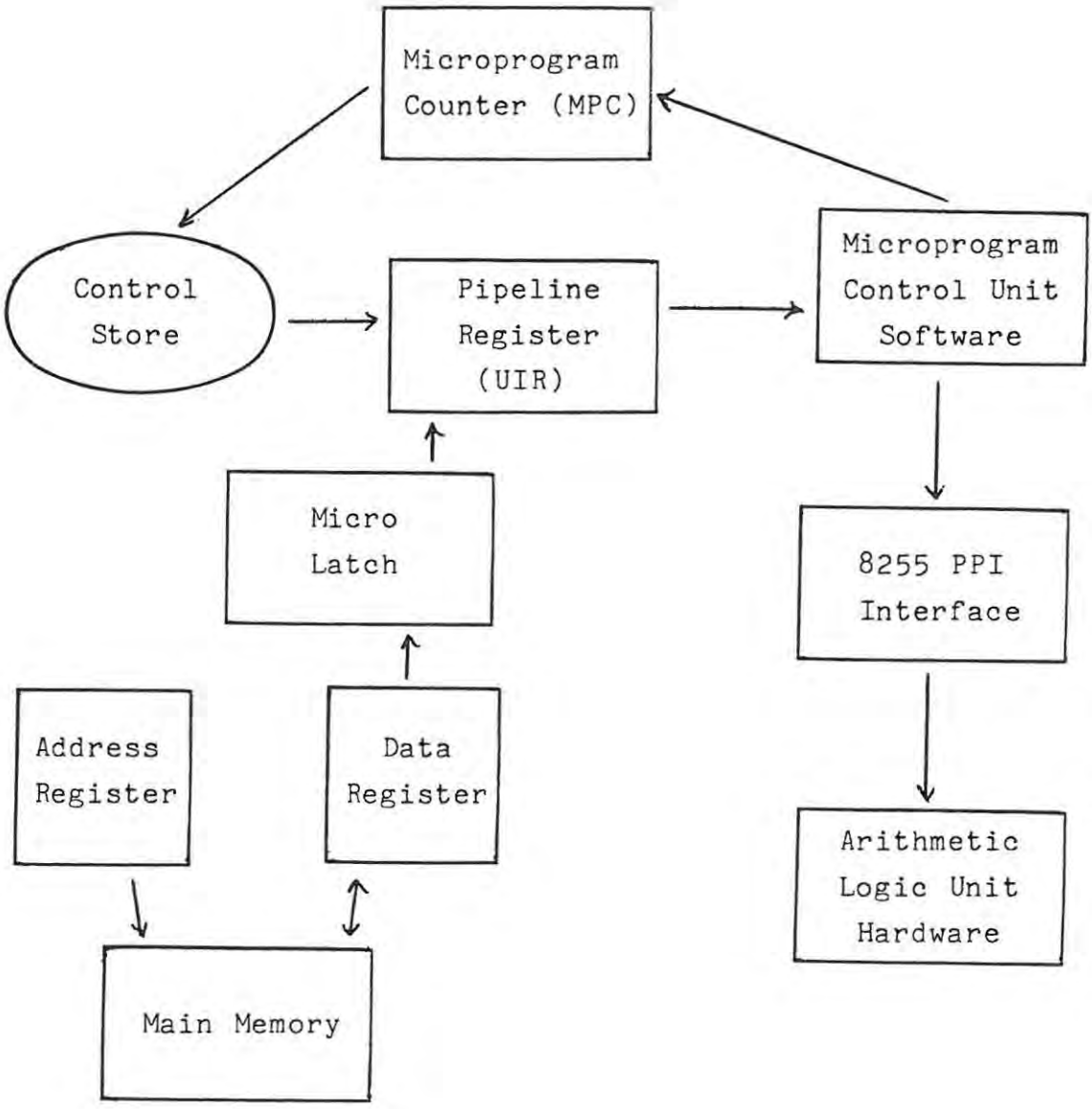


Figure 1.7
CONTROL PATH SCHEMATIC OF THE TRIPLE-M MACHINE

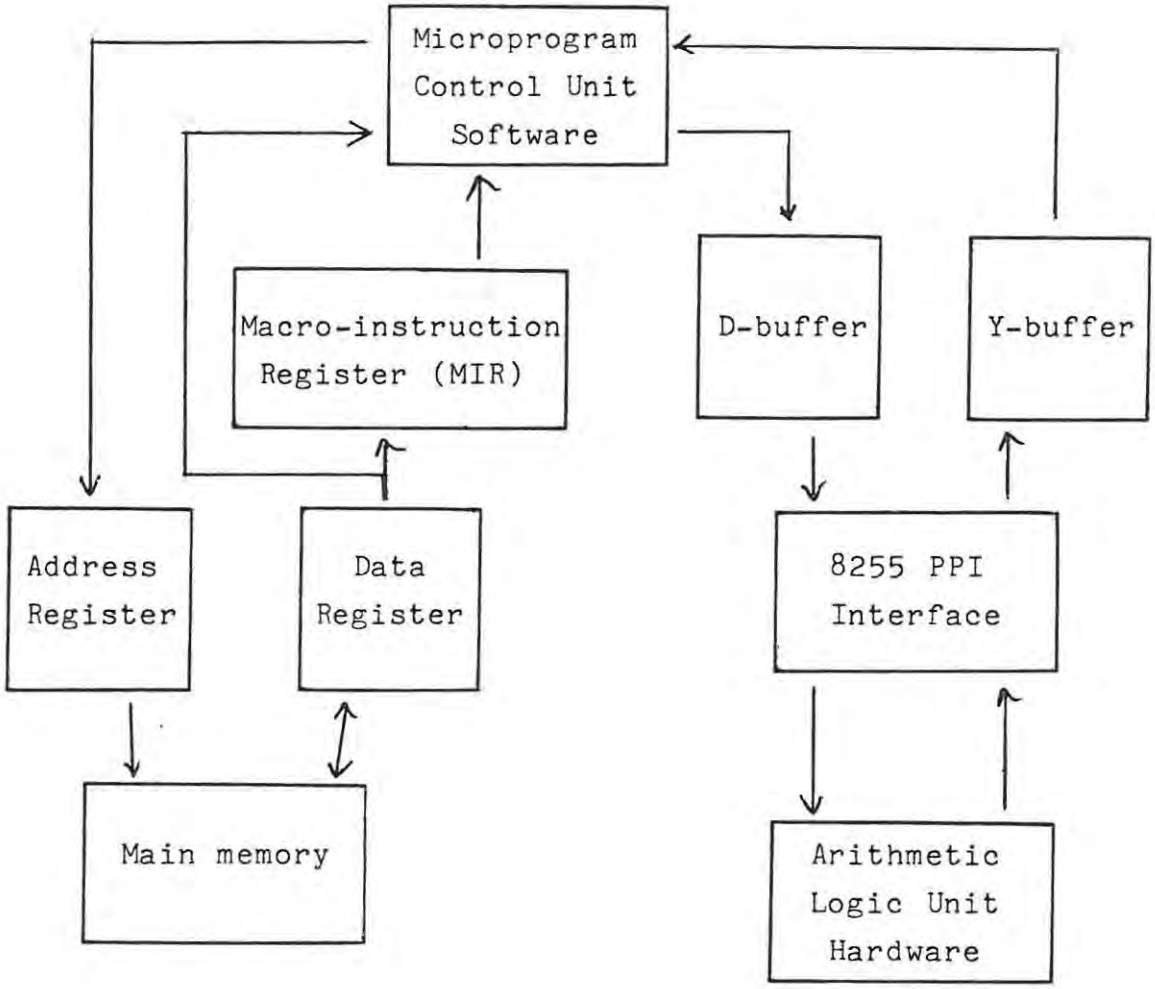


Figure 1.8
DATA PATH SCHEMATIC OF THE TRIPLE-M MACHINE

If the value in the Data Register is not an instruction, then it is either loaded into the Micro-Latch or, more usually, into the D-buffer.

The D-buffer is the input buffer to the ALU. Once the ALU has performed the required operation, the result (if any) can be picked up on the Y-buffer. This Y-buffer value may then either be loaded into the Address Register or the Data Register. Figure 1.8 is a schematic diagram of the data paths of the Triple-M machine.

Although the Triple-M machine is a unique microprogrammable processor, the procedures in the MCU software have been designed to enable other types of hardware ALU's to be used with the system. Such an implementation would involve a certain amount of programming and the construction of the ALU itself. The procedures in the MCU software that control the sequencing functions could also be changed to simulate other controller/sequencers.

The present version of the MCU requires the first 24 bits of the microinstruction to be reserved for its functions. These functions have been designed with the Triple-M machine in mind but were based on those provided by Advanced Micro Devices' Am2909 and Am2910 controller/sequencers. The Triple-M controller functions are tabulated below and one should refer to figures 1.7 and 1.8 to clarify the 'actions' listed.

A field to hold a jump address or a constant value:

I1 to I12	Action	Mnemonic
Binary address	Specified by I14 and I15	None

Increment the Micro-Program Counter:

I13	Action	Mnemonic
0	Increment	INCR
1	No increment	NOINCR

Branching:

I14 I15	Action	Mnemonic
00	No Branch	NOBRN
01	Unconditional	UNCOND
10	Conditional	COND
11	Constant field	CONST

Memory access:

I16 I17 I18	Action	Mnemonic
000	No memory access	NMA
001	Data register -> D-buffer	DR-DB
010	Y-buffer -> Address register	YB-AR
011	Y-buffer -> Data register	YB-DR
100	Data register -> Micro Latch	DR-UIR
101	Micro Latch -> UIR	MPC-AR
110	Data register -> MIR and decode	DR-MIR
111	Data register -> Address register	DR-AR

Clock field:

I19	Action	Mnemonic
0	Don't send a clock pulse	CN
1	Send a clock pulse	CY

Free:

I20 to I24	Action	Mnemonic
Not used	None	None

The rest of the microinstruction is available for control of the ALU hardware. The Triple-M machine's ALU, constructed around four Am2901 processor slices, requires a further 32 control bits giving a total microinstruction width of 56 bits. The layout of this microinstruction format is given in figure 1.9. This format was decided upon with the aid of the tables of functions that accompanied the Am2900 family of bit-slice chips. An abridged copy of these tables is given in appendix F for completeness.

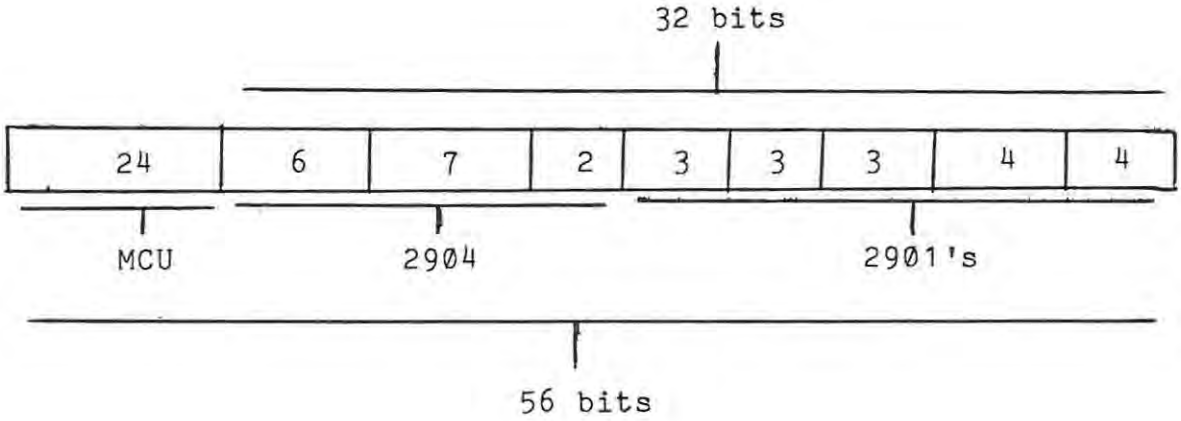


Figure 1.9
THE MICROINSTRUCTION FIELDS

1.6 : The Arithmetic Logic Unit hardware.

The ALU hardware is constructed using components from AMD's Am2900 bit-slice family. The heart of the ALU consists of 4 Am2901B bipolar microprocessor slice. Each of them is four bits wide giving a 16 bit wide processor. An Am2902A high-speed look-ahead carry generator is used to provide the look ahead over the 16-bit word. An Am2904 status and shift control unit replaces most of the other medium scale

integration circuits that are normally required around the ALU.

The Am2901B slice is designed as a high speed cascadable element intended for use in CPU's and for other applications. The microinstruction flexibility should allow efficient emulation of almost any digital computing machine. The Am2902A look-ahead carry generator has been designed for use with up to four pairs of carry propagate and carry generate signals and is therefore ideally suited for use with use

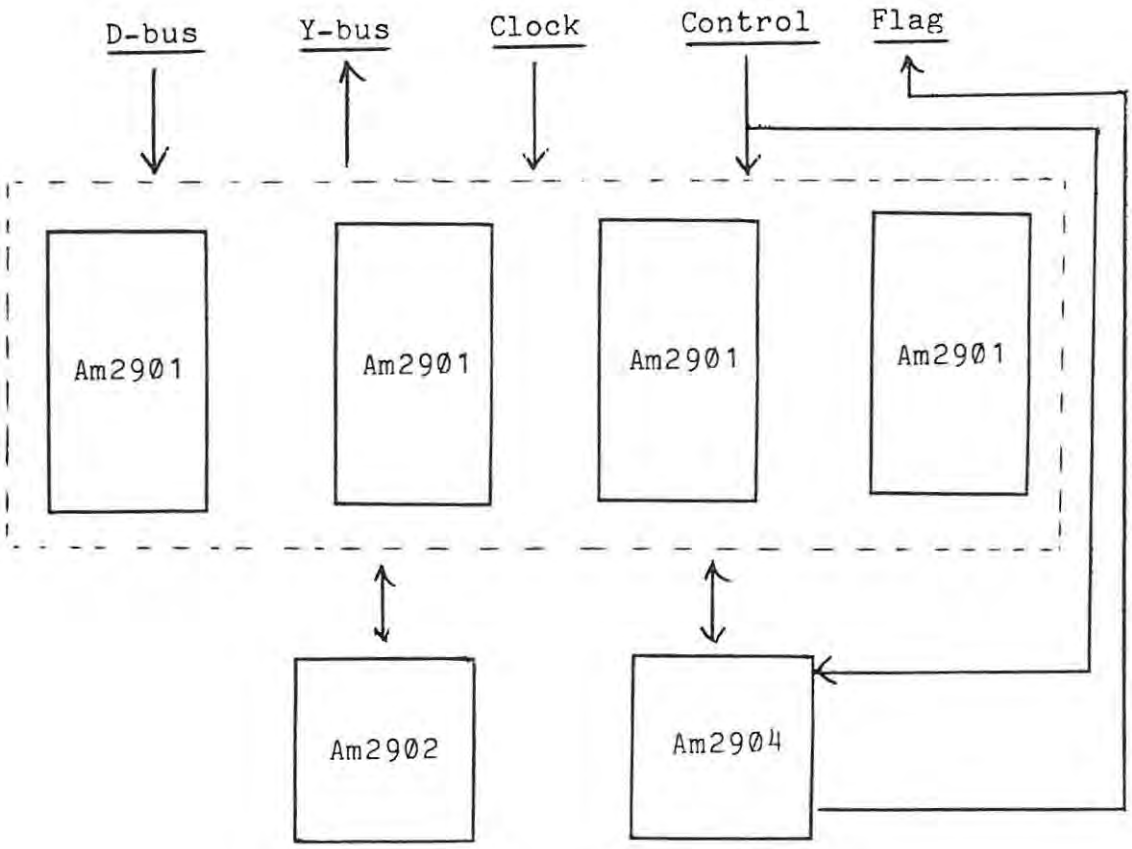


Figure 1.10
SCHEMATIC DIAGRAM OF THE ALU

with the four 2901 ALU slices. The Am2904 is designed to perform all the miscellaneous functions which are usually performed in MSI (medium scale integrated circuits) around the ALU. These include the generation of the carry-in signal to the ALU, the various types of shift linkages and the storage and testing of the ALU status flags. There are two separate status flag registers, a microstatus register and a machine status register [ADV].

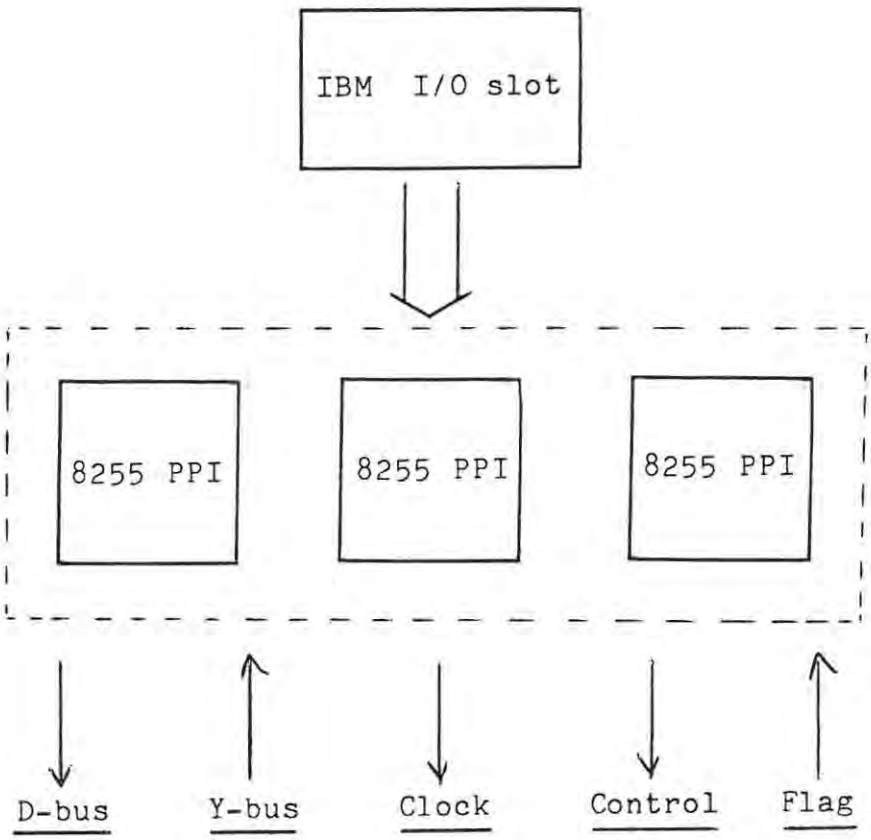


Figure 1.11
CONNECTIONS BETWEEN THE ALU AND MCU

These chips are connected in the standard way suggested in each of the chip specifications and figure 1.10 represents this basic layout. Detailed circuit diagrams are given in appendix H.

Three Intel 8255A Programmable Peripheral Interface (PPI) chips were used to connect the ALU hardware to the MCU software. The Intel 8255A's are general purpose programmable Input/Output devices designed for use with Intel microprocessors [GOL].

The various connections between the ALU and the MCU are shown schematically in figure 1.11. The PPI's are accessed at addresses $3E0 to $3EB which are free in the I/O address space [IBM]. These address are decoded using a NAND gate and a few NOT, NOR and OR gates. The decoding circuitry is given in appendix H along with more detailed circuit diagrams of the interface hardware.

Figure 1.12
THE BREAD BOARD ALU

The ALU and interface hardware was initially built on 5 experimental bread boards giving rise to the 'birds nest' shown in figure 1.12. The circuit as shown actually worked very well but proved unsatisfactory as the connections were so easily disturbed.

The final circuit was wire wrapped, using a 'just wrap' wire wrapping tool, on an IBM prototyping board. This proved far neater and more reliable. Figures 1.13 shows the ALU board which easily plugs into any one of the I/O slots in the IBM's mother board.

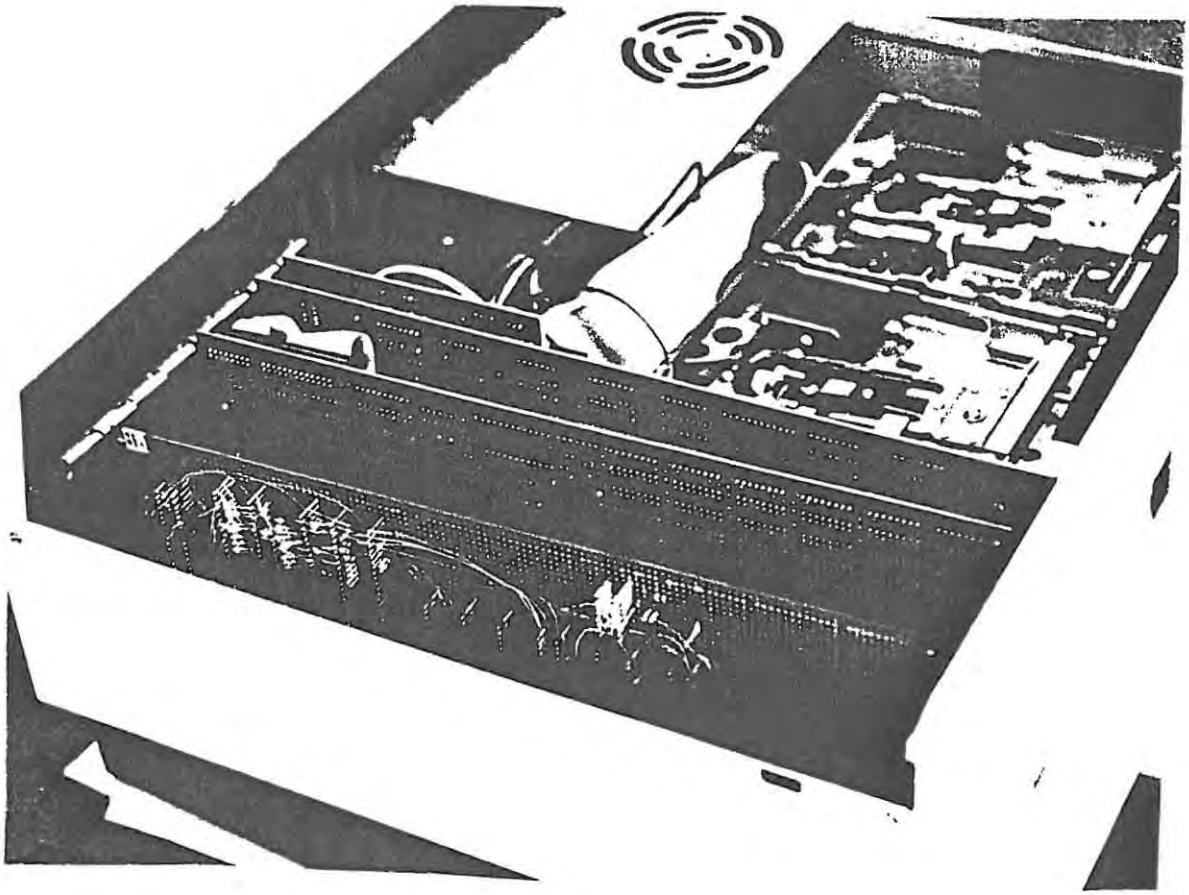


Figure 1.13
THE WIRE WRAPPED PROTOTYPING BOARD

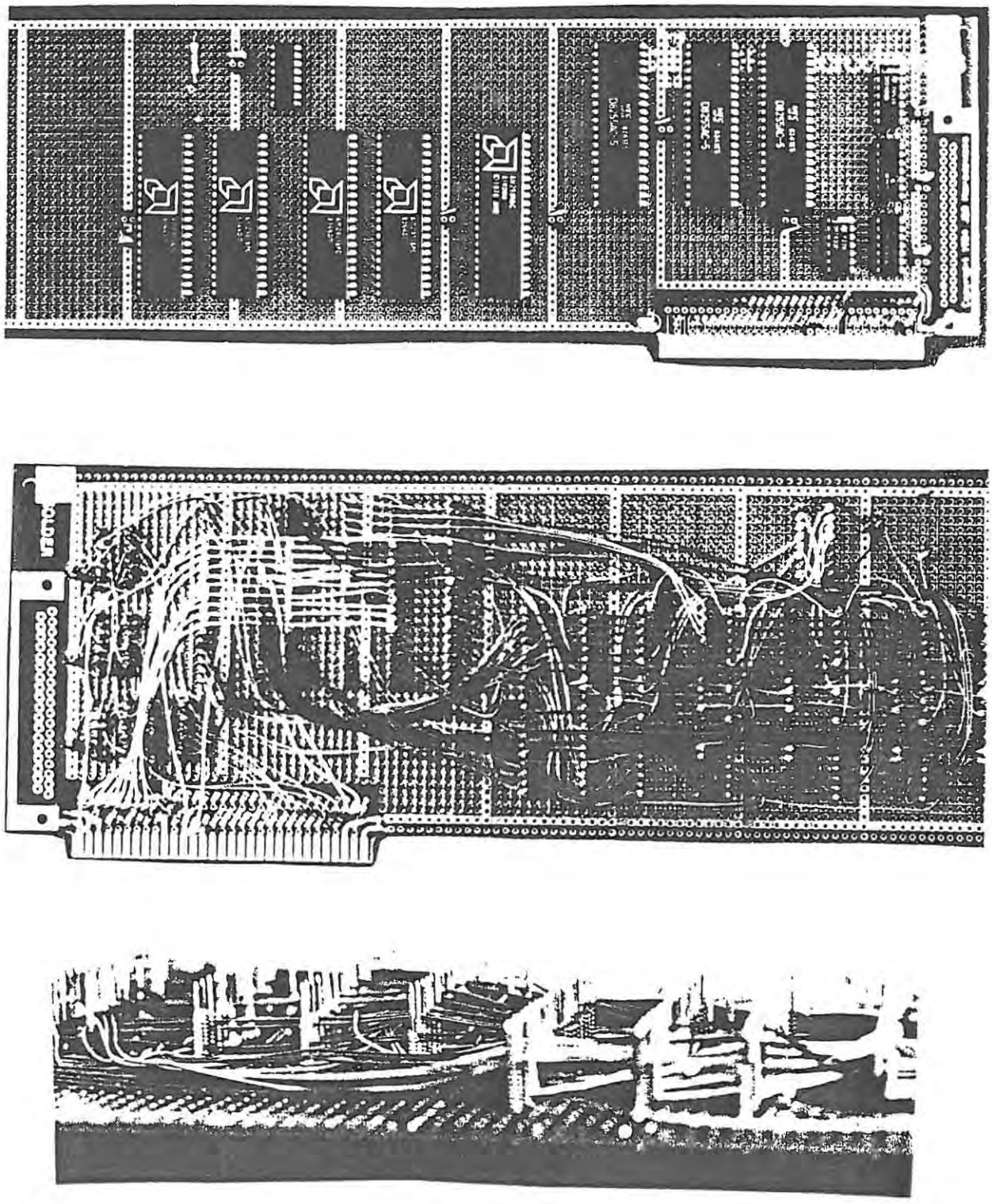


Figure 1.13
THE WIRE WRAPPED PROTOTYPING BOARD

Figure 1.14 gives a break down of how the I/O lines of the three PPI's were assigned. There are three 8-bit wide ports A, B and C for each of the PPI's. Ports 1A, 1B, 1C and 2A give the required 32 bits needed for the hardware control signals. The lower nibble of port 2C is an input

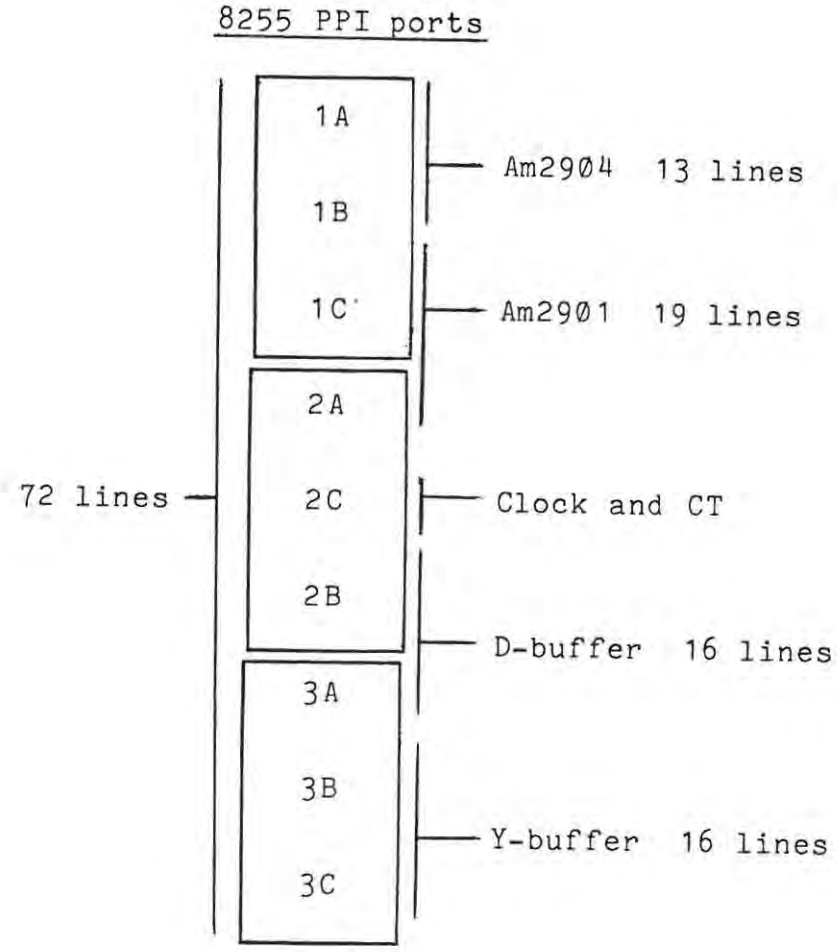


Figure 1.14
PORT ASSIGNMENT DIAGRAM

buffer used to read the status flag return signal from the Am2904. The upper nibble is used to send the clock pulse to the hardware. Ports 2B and 3A are used for the 16-bit wide D-buffer and ports 3B and 3C for the 16-bit Y-buffer.

The software that controls the PPI's is described in the section on the MCU emulator.

1.7 : The Microprogram Control Unit emulator.

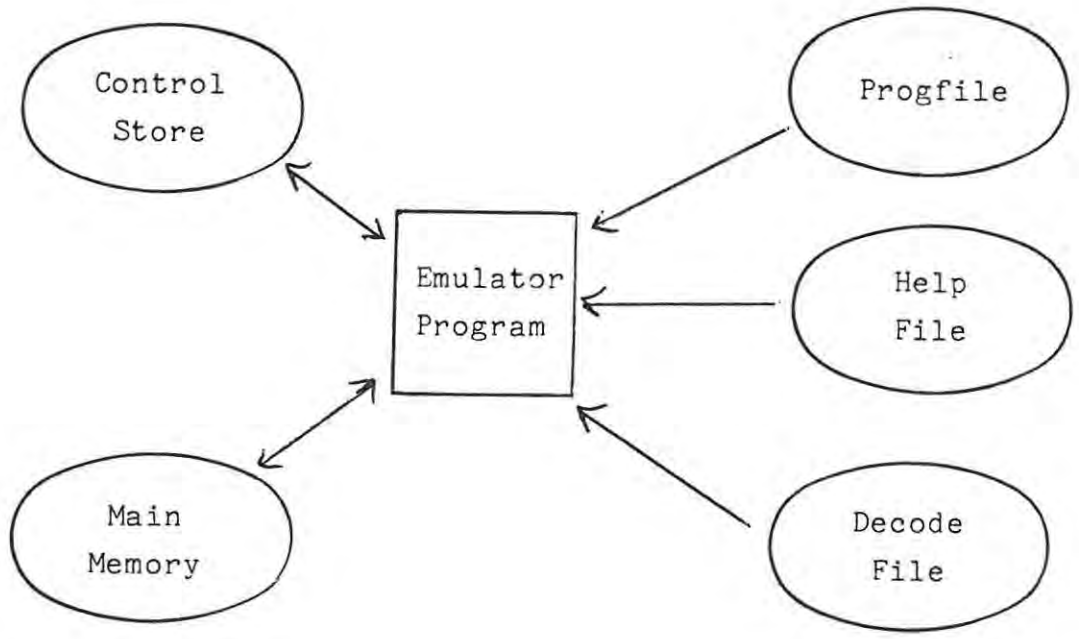


Figure 1.15
THE EMULATOR PROGRAM FILES

The Emulator is the software that produces the effect of the microprogram control unit (MCU). It also contains the routines that allow easy debugging and controlled execution of the microprogram.

Microinstructions are loaded from the control store file into the pipeline register (UIR) and executed. The emulator assumes that the macro-program to be run has already been assembled, and is resident in a file called the progfile. The program instructions or opcodes are read from this file into main memory. Main memory is a file of integers simulating a 16 bit wide memory. Each address used to access the file therefore locates one 16 bit memory location. The decode file, set up by the Micro2 program, is used to find the micro-routine in control store for each macro-instruction or opcode. Figure 1.15 shows the basic layout of the files used.

The Emulator program is based on the following algorithm:

```
Begin
   Include the helpfile if requested
   Assign the control store and decode files
   Ask whether the user wishes to simulate the hardware
   Ask for the name of the macro-program to be run
   Load the macro-program into the 'main memory'
   While it is not the end of the session do
   Begin
      Write out the main menu
      Then either  run the program
                Or  single step through the program
                Or  edit the micro-codes
                Or  trace the execution
                Or  look at specific register values
                Or  change the D-bus value
                Or  change the MPC
```

```
                Or  peek memory locations
                Or  ask for help
                Or  terminate the session
        End
    Close all the assigned files
    End.
```

A few preliminary functions have to be performed before the main program may commence. The helpfile facility may be included at the request of the user. The control store and the decode files have to be linked to their real file names. The user may also choose whether to simulate the hardware or not, and thereafter provide the name of the macro-program to be run. The system loads the macroprogram into the main memory file, starting at address 2000.

The main program allows the user to select one of the options given in the main menu. The trace function may either be active or inactive. The address at which the macro-program begins should be entered using the 'D-bus' command so that the program counter may be initialised. In most cases this will be 2000 as this is where the loader puts the macroprogram in main memory. A 'startup' routine in the microprogram should be designed such that it takes the value on the 'D-bus' and places it in a hardware register that is to be used as the program counter. This proved to be the easiest and most logical way of initialising or updating the program counter. The value of the Micro-Program Counter (MPC) should then be changed to point to the 'startup' routine in the microprogram. It is possible to either single step through the microinstructions or to let the program run without intervention. The user may edit the microcode by changing one microinstruction at a time. These microinstructions may be executed again by changing the value of the MPC to point to the 'startup' routine and putting the appropriate program counter value on the 'D-bus'. The emulator also allows the user to look at specific register

values which proves very useful when debugging. A peek option has been included to enable the user to examine specific main memory locations. He is thus able to check that macroprogram values have been stored correctly. The memory locations may only be viewed and cannot be updated using this option.

The structure of the control store and the decode files has already been discussed in the section on the Micro2 program. The progfile is a file of integer values in accordance with a 16-bit wide memory.

The helpfile facility is available when input is required from the user and has been implemented in the same way as described in the section on the Micro1 program.

More should be said here about the code that performs the function of the microprogram control unit. There are five fields in the microinstruction that control the MCU. The first is a jump field that may either hold a jump address in the microprogram or it may hold a constant value. There is an increment field that indicates whether or not the Micro-Program Counter (MPC) is to be incremented. A branching field that determines the type of action to be taken on the jump address. A memory access field that controls the Address Register (AR), Data Register (DR), D-buffer, Y-buffer and the Macro-Instruction Register (MIR). Finally there is a clock field that indicates whether or not the hardware is to be accessed. Tables of these functions are given in appendix F.

To execute one microinstruction the program isolates each of these micro-orders from that microinstruction and then follows the algorithm below: (The reader may refer to figures 1.7 and 1.8)

```
Begin (* execute one microinstruction *)
   If the branching field is
      '00' then do nothing
      '01' then put the jump address into the MPC and load the
           microinstruction at this address into the Micro-Instruction
           Register (UIR)
      '10' then if the tested flag is TRUE do the same as '01' above
           else increment the MPC by 1 and load the microinstruction
           at that address into the UIR
      '11' then load the constant from the jump address field into the
           D-buffer.
   If the memory access field is
      '000' then if the clock field is 1 then send the control signals
           to the hardware; this operation is performed in the
           functions below where the word 'clockfield' is written
      '001' then read from main memory at the address in the address
           register and put the data from the data register onto the
           D-buffer and clockfield
      '010' then clockfield and put the value from the Y-buffer into
           the address register
      '011' then clockfield and put the value from the Y-buffer into
           the data register and store this data at the address
           found in the address register
      '100' then put the value from the data register into the Micro
           Latch at the position given by the value in the jump
           address field, and clockfield
      '101' then put the value in the Micro Latch into the UIR and
           clockfield
      '110' then put the data register value into the Macro-
           instruction register and decode the macro-instruction to
           give an address in the microprogram; fetch that
           microinstruction and put it into the UIR; clockfield
      '111' then put the data register value into the address register
```

```
            and clockfield.
   If the increment the Micro-Program Counter (MPC) is
      'Ø'  then increment the MPC and put the microinstruction at  that
           address into the UIR
      '1'  then do nothing.
End (* execute one microinstruction *)
```

Note that if the system is in 'single step' mode, it will wait until a command is given before executing the next microinstruction.

Sending the control signals to the hardware is performed by a procedure containing the interface code that drives the Intel 8255 PPI's. The D-buffer value is also sent and the clock pulse then tells the hardware to execute those microcodes on the control lines. After the microinstruction has been executed, the Y-buffer value is read in and the test flag result (CT) from the Am29Ø4 is obtained. The following algorithm shows how these events are scheduled.

```
Begin (* interface code *)
   Pull the clock line high
   Send the ALU control signals
   Send the D-buffer value
   Bring the clock line low
   Read the CT test flag result
   Pick up the Y-buffer value
   Pull the clock high
End (* interface code *)
```

The ALU control signals and the D-buffer value are held in the output buffers of the 8255's. The falling edge of the clock signals the ALU to perform the desired functions. The ALU performs the operation so quickly that the test flag result (CT) and the Y-buffer value are ready long before the program issues the next command to read them in. This

effectively means that no handshaking lines are needed to assist the communication.

Two execution modes have been supplied, namely the 'single step' and the 'run' modes. In single step mode, the user is presented with debugging information on the screen. The microinstructions are listed as they are executed and various register values are given if the trace mode is active. The user may also employ the 'look' option the view the Am2901 register array values. At any point during the execution, control may be swapped to the main menu where other options may be chosen before returning to the single step mode to continue the execution.

In run mode, the user has no access to debugging facilities. (If the trace mode is active, the screen will be corrupted and the user will not be able to clearly see any I/O that the macroprogram is attempting.) Upon initiating the execution, the screen is cleared and the user is presented with the normal "external" view of the machine. He must now perform the usual input/output functions to monitor the program execution. The option does exist to hold the space bar down in order to gain a sneak re-entry to the main menu.

The first 2000 memory locations in the 'main memory' (addresses 0-1999) are mapped onto the active screen. Storing characters in these memory locations causes them to be echoed on the screen. If one of these memory locations is read by the program, then the system waits for the user to enter a key from the keyboard. This system provides the user with a crude form of input/output, without the need for explicit I/O drivers in the Triple-M machine. Using these ideas the user should easily be able to perform normal I/O functions.

A listing of the Emulator program code is given in appendix C.1 and C.2.

1.8 : The ALU Simulator.

The Simulator forms a small part of the Emulator software and is included to allow testing of the microprogram to be done independently of the hardware. The Simulator therefore produces the effect of the arithmetic and logic unit hardware.

All the ALU functions are provided, except the complex shift linkage routines made possible by using AMD's Am2904 in the hardware. Most of the functions provided by this chip would not be necessary in a normal programming environment.

The corresponding portion of the microinstruction, in this case the last 32 bits, is passed to the Simulator, which isolates the various micro-orders and performs the associated functions. The controller/sequencer is not aware of whether the micro-orders are being executed by the ALU hardware or the ALU Simulator because the calling routine passes the same 32 bits in either case. Depending on whether the hardware is to be used or not, the bits are either sent to the hardware or to the Simulator. The Simulator is totally independent of the controller/sequencer, this being a desirable feature of a true simulator. The Simulator's registers are implemented as global integer variables, while the ALU status flags are of type Boolean.

Of particular interest is the implementation of the overflow flag and the carry flag. These can be set by looking at the values of the arguments presented to the ALU and the resulting answer. If arguments of opposite signs are added then there is no overflow, but if numbers of the same sign are added and the answer is of the opposite sign, then an overflow has occurred. The carry flag requires a more complicated algorithm to compute. If two positive numbers are added then there can be no carry but if two negative number are added there is always a

carry produced. Adding operands of opposite signs gives a carry if the answer is positive and gives no carry if the answer is negative. The same rules apply for subtraction. The two's complement of the second argument is added to the first argument and the same checks made.

The Simulator program code is listed in appendix C.2.

2 : EXAMPLE.

2.1 : Introduction.

SYSMW as a whole is perhaps best seen in terms of a practical example. It was decided that assembly language instructions would best illustrate all aspects of the system.

The two principal types of computer architectures in widespread use are the stack type architectures and the register architectures [AME]. In the example described below, a minimum set of machine language instructions was developed to emulate a single register architecture.

An early step in all design implementations is the creation of the target machine description. In the case of this example the target machine was SYSMW's Triple-M machine. The Micro1 program was used initially to define all the format templates for the micro-orders, and later to define the micro-order mnemonics. Listings of the resultant format and definition files are given in appendix G.1 and G.1.

The format and definition files were used to create the table file which contains the complete machine definition. Although it is not normally necessary for the user to be able to list this file, a copy has been included in appendix G.3 to afford the reader a view of its structure.

The Micro2 program was employed to create the necessary microcode segments to emulate the desired macroinstructions.

All computer systems require some form of boot routine either to start the operating system or to initialise various key registers. In this example, the 'startup' routine is a microcode sequence that initialises the value of the program counter. Of the sixteen internal registers in

the Am2901 based ALU, the first was chosen as the program counter, while the second was used as the general accumulator. The startup routine reads as follows:

```
Micro-routine: startup
Micro-codes:
Breg #0000 & DZ & OR & RAMF & CY ;assume PC is on the D-bus
NOP & JUMPADDR #fetch & NOINCR & UNCOND & KF
```

The first microinstruction assumes that the value of the program counter is present on the D-bus, and this value is loaded into the register at address '0000'. The user must ensure that the desired program counter is loaded onto the D-bus using the commands provided by the Emulator program. The second microinstruction performs a jump to the 'fetch' routine which is the first of such calls in the long line of fetch-execute cycles.

The fetch micro-routine gets the next opcode from main memory and it is shown below:

```
Micro-routine: fetch
Micro-codes:
Areg #0000 & Breg #0000 & RAMA & ZB & ADD & CO=1 & I-U & YB-AR & CY
NOP & DR-MIR & NOINCR & KF
```

The first microinstruction highlights the ability of horizontal designs to perform various functions in one microinstruction. The program counter is put out on the Y-bus and then transferred to the Address Register. At the same time, the program counter is incremented by one and returned to its location. The second microinstructions reads from main memory at the address given in the Address Register and puts this Data Register value into the Macro Instruction Register (MIR). This instruction is then decoded (using the information in the decode file)

to give an address in the microprogram where the microcode segment for that instruction resides. This address is put into the Micro Program Counter (MPC) which simulates a jump to that address in the control store.

In all of the microinstructions developed for this system, there is either a 'I-U' or 'I-M' or 'KF' micro-order. The I-U instruction loads the test flags (zero, carry, sign, and overflow flags), set by the ALU operation, into the Micro Status Register, while the I-M instruction loads the flags into the Macro Status Register. Only the Macro Status Register flags should be visible to the macroprogram and it is these flags that are tested in the conditional branch macroinstructions. The KF micro-order must be included if one of the load flag instructions is not used, to keep the Micro and Macro Status Register flag values from being overwritten.

The 'CY' micro-order is used to indicate to the system that the ALU is involved in that microinstruction. If CY is not included then the ALU will not be accessed. This has been implemented to increase the efficiency of the system so that time is not wasted in sending signals to the ALU via the involved interface routines, unless this is necessary. The user should therefore ensure that the CY micro-order is included if there are other micro-orders in that microinstruction that require the ALU.

The user may employ the branching facilities, made available by the system, not only to jump to the fetch routine at the end of every microcode sequence, but also to execute a sequence of microinstructions in a procedural way.

It is also possible to have looping structures within the defined micro-routines. The example below decrements a register until its value is zero. This example could be used as a delay micro-routine.

```
Macro-instruction  :DELAY
Hexadecimal opcode :FF
Micro-codes:
:LAB1 Breg #1111 & ZB & SUBR & NOP & I-U & CY
NOP & u5 & CY & KF
NOP & JUMPADDR #LAB1 & NOINCR & COND & KF
NOP & JUMPADDR #fetch & NOINCR & UNCOND & KF
```

Note that the Micro Status Register is used to store the flags, and it is the 'micro zero flag' that is tested in the conditional jump. If the zero flag is not set, the control returns to the first microinstruction and the register is again decremented. Once the register is zero, the last microinstruction is executed, being the usual jump to the fetch micro-routine.

In all cases labels have to be preceded by a ':'. A ';' denotes the start of a comment field.

2.2 : The Machine Language Instructions.

The macro-instructions developed for the single register type architecture are listed below:

```
LDA#  ;load an immediate value into the accumulator
LDA   ;load the acc from the address in the next memory location
STA   ;store the acc at the address in the next memory location
ADD#  ;add an immediate value to the acc
ADD   ;add the value at the given address into the acc
SUB#  ;subtract an immediate value from the acc
SUB   ;subtract the value at the given address from the acc
```

```
NOT     ;invert the accumulator
AND#    ;logical AND of immediate value with the acc
AND     ;logical AND of the value at the given address with the acc
OR#     ;logical OR of immediate value with the acc
OR      ;logical OR of the value at the given address with the acc
SHL     ;arithmetic shift left of acc
SHR     ;arithmetic shift right of acc
JMP     ;unconditional jump
JMZ     ;jump if zero flag set
JMNZ    ;jump if zero flag clear
JMNEG   ;jump if negative flag set
JMOVR   ;jump if overflow flag set
CMP#    ;compare immediate value with acc, set flags, acc unchanged
CMP     ;compare the value at the given address with the acc
```

A full list of these instructions and their associated microinstruction segments is given in appendix G.4. Most of these instructions require a further memory access to fetch an operand. In these cases the microinstruction shown below is included as the first in the sequence.

```
Areg #0000 & Breg & RAMA & ZB & ADD & CO=1 & I-U & YB-AR & CY
```

This is identical to the first microinstruction in the fetch micro-routine, and accesses the memory location following the opcode while incrementing and storing the program counter.

One further macroinstruction that has not been mentioned is the 'INL' instruction. This is an extension to the normal assembly language type instructions and its concept is motivated in Part 1. Briefly, this type of instruction could prove very useful if the microprogram has been fixed in ROM and the user then decides that he wishes to implement a specific operation in microcode. These exotic microinstructions may then be included in the macroprogram as 'inline' microcode.

The 'INL' macroinstruction and its associated microcodes are given below:

```
Macro-instruction:  INL
Hexadecimal opcode: Ø1
Micro-codes
Areg #ØØØØ & Breg #ØØØØ & RAMA & ZB & ADD & CO=1 & I-U & YB-AR & CY
DR-UIR & JUMPADDR #ØØØØØØØØØØØØ & KF ;DR-LAT
Areg #ØØØØ & Breg #ØØØØ & RAMA & ZB & ADD & CO=1 & I-U & YB-AR & CY
DR-UIR & JUMPADDR #ØØØØØØØØØØØ1 & KF ;DR-LAT
Areg #ØØØØ & Breg #ØØØØ & RAMA & ZB & ADD & CO=1 & I-U & YB-AR & CY
DR-UIR & JUMPADDR #ØØØØØØØØØØ1Ø & KF ;DR-LAT
Areg #ØØØØ & Breg #ØØØØ & RAMA & ZB & ADD & CO=1 & I-U & YB-AR & CY
DR-UIR & JUMPADDR #ØØØØØØØØØØ11 & KF ;DR-LAT
MPC-AR & KF & NOINCR ;LAT-UIR
NOP & JUMPADDR #fetch & NOINCR & UNCOND & KF
```

Figure 2.1 gives a closer view of the parts involved in the implementation of this extension.

The first eight microinstructions, in the INL description, are responsible for loading the inline microinstruction data from main memory into the Micro Latch. The position in the latch into which the data is to be loaded is given by the value in the 'jumpaddress' field. The ninth microinstruction loads the Micro-Instruction Register (UIR) from the Micro Latch and does not increment the MPC (microprogram counter). The next microinstruction to be executed is therefore the 'inline' one. This inline microinstruction should allow the MPC to increment and fetch the next instruction in control store which will be the tenth one in the sequence shown above. This microinstruction performs the usual jump to the 'fetch' micro-routine that occurs at the end of all the microcode segments.

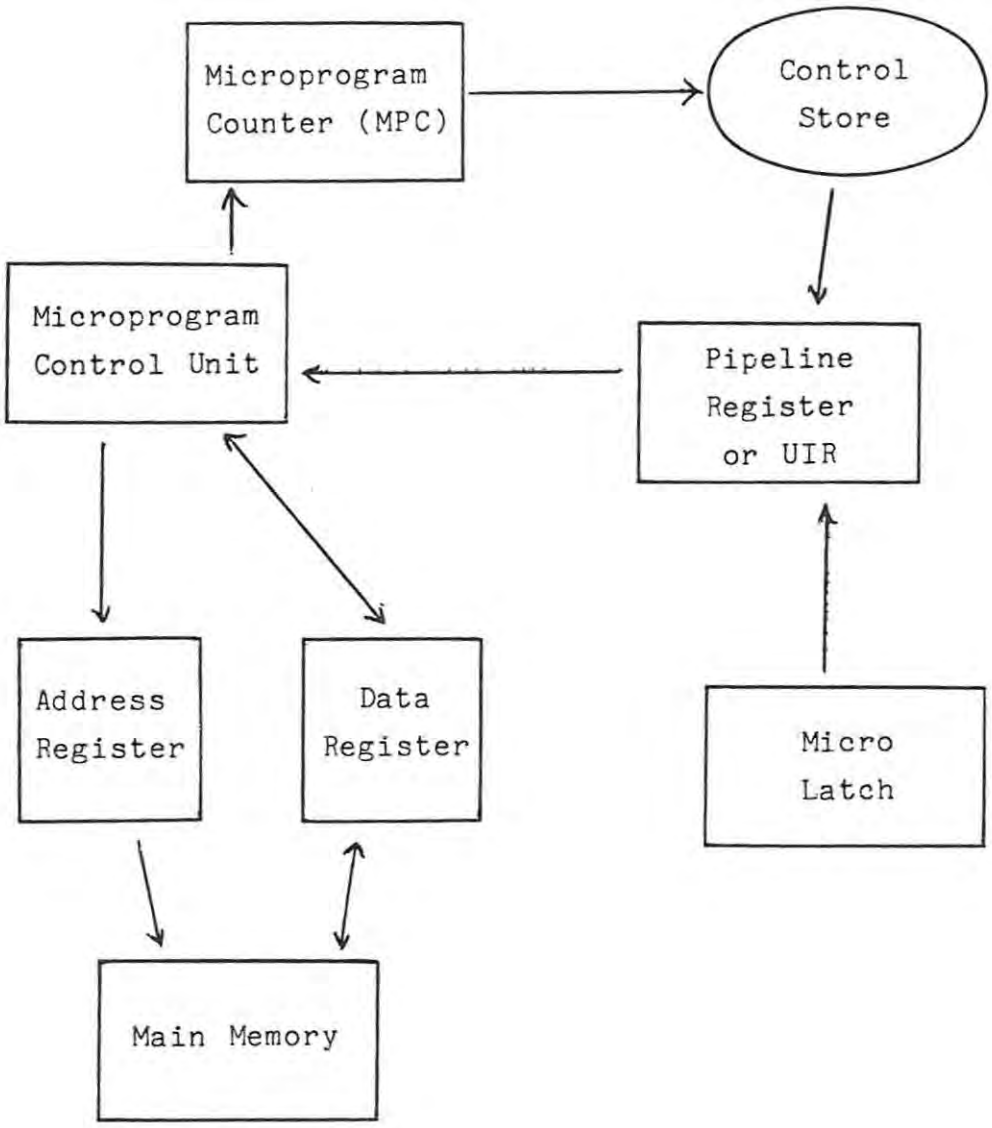


Figure 2.1
THE INLINE INSTRUCTION DATA PATHS

Listings of the macroinstruction, decode, instruction and control store files, for this example are given in appendices G.4 through G.7.

2.3 : The Mini Assembler Program.

A simple assembler was written to facilitate the writing of the macroprograms. Strictly speaking, it does not form part of SYSMW, as it has very little to do with the design or execution of microcode.

The assembler, called AssemMW, is a two pass assembler that accepts as input a text file containing the macroprogram mnemonics. It produces an output file containing the machine instruction opcodes and their associated operands. The decode file from the Micro2 program is used to provide the opcode values for the mnemonics in the program to be assembled. Figure 2.2 gives a general layout of the files used.

The algorithm on which the program is based is as follows:

```
Begin
   Enter the name of the control store file
   Enter the name of the source program to be assembled
   While it is not the end of the session do
      Print out the main menu
      Then either  edit the object code file
                Or  assemble the source code
                Or  print the object code file
                Or  ask for help
                Or  terminate the session
   Close all the assigned files
End.
```

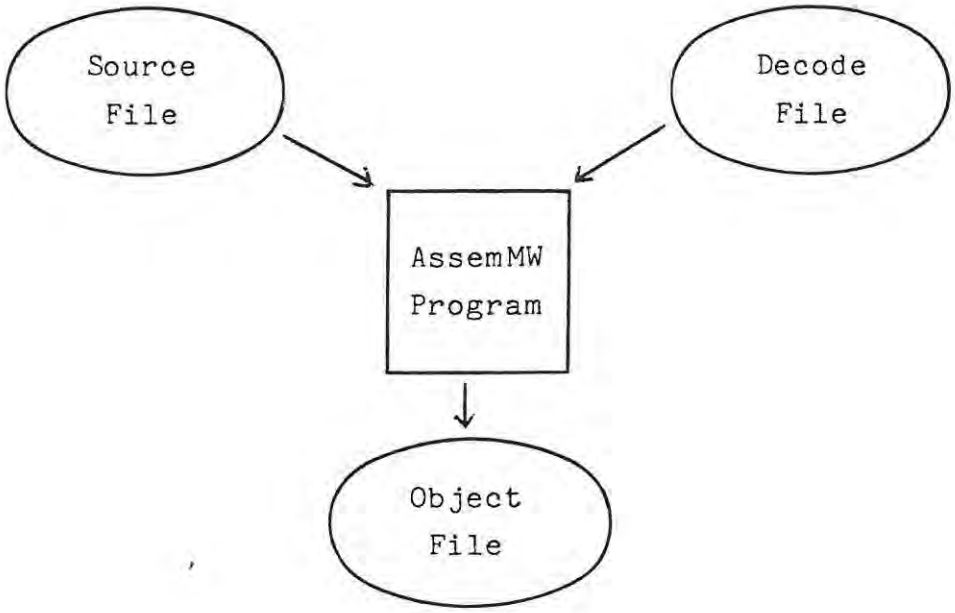


Figure 2.2
THE ASSEMMW PROGRAM FILES

Although the user is requested to enter the name of the control store file, it is actually the decode file of the same name, that is used by the program. The source program to be assembled has to have been previously entered and must be assigned to a file with the extension '.SCE'. The assembled version of the program will be written to an object code file with the extension '.OBJ'.

The user is able to select one of the options available in the main menu. An existing object code file may be edited thus allowing the user to put values into the main memory when this program is loaded. The

assemble command initiates the two pass assembler that converts the source code into the object code instructions. The print option lists the object file to either the screen or the printer.

A listing of an example input file is given below:

```
;Program Test
        LDA# 2      ;load 2 into acc
:LAB1   SUB# 1      ;subtract 1
        JMZ LAB2    ;jump if zero to lab2
        JMP LAB1    ;jump to lab1
:LAB2   INL         ;an inline microinstruction follows
        Ø
        8432
        1470
        256
;End Test
```

As in the case of microinstruction definitions, labels have to be preceded by a ':' and a ';' denotes the start of a comment field. The inline microinstruction shown is one that complements the accumulator and the decimal values given are converted into the microinstruction below:

```
ØØØØØØØØØØØØØØØØØØØ1ØØØØØ1111ØØØØØØØØØ1Ø11Ø1111ØØØØØØØ1
```

The main memory is loaded with the assembled version of the program and for the example above, the memory locations would read as follows:

Memory address	Value
2000	16
2001	2
2002	64
2003	1
2004	97
2005	2008
2006	96
2007	2002
2008	1
2009	0
2010	8432
2011	1470
2012	256

Note that the addresses start at 2000, as this is where the Emulator program will load all object code segments. The jump addresses are therefore also calculated with an offset of 2000.

A listing of the AssemMW program code is given in appendix D.

3 : A USER'S GUIDE.

3.1 : Introduction.

This user's guide is divided into four main sections. The first two concern the programs involved with microprogram development. The third covers the Emulator and the Triple-M machine's associated functions while the last briefly discusses the AssemMW program, a macroprogram assembler.

Section 3.2 discusses the operating procedures of the Micro1 program which deals with the definition of the target microprogrammable machine. Section 3.3 deals with Micro2, the microinstruction definition program. Section 3.4 describes the operation of the Emulator program which forms part of the Triple-M microprogrammable machine. Finally section 3.5 discusses the AssemMW program which is a simple assembler used to compile the example language developed to test the functions of the microprogram development system and the Triple-M machine.

3.2 : Micro1 Program.

After the start of the Micro1 session, the user is required to supply the name of a workfile and then has to decide on the inclusion of the help messages.

For convenience, the work file name is associated with the three related format, definition and table files. Each of these is assigned a different extension by the program. For the format file this is '.FMT', for the definition file it is '.DEF' and for the table file, '.TAB'.

If the help facility has not been included, only the help message

number is listed and not the help message itself. The user is required to consult the helpfile listings to find the corresponding help message. The information contained in the help messages appears in this manual and is listed in appendix E with its corresponding numerical index. The program response time to a help request is better if the help messages are excluded.

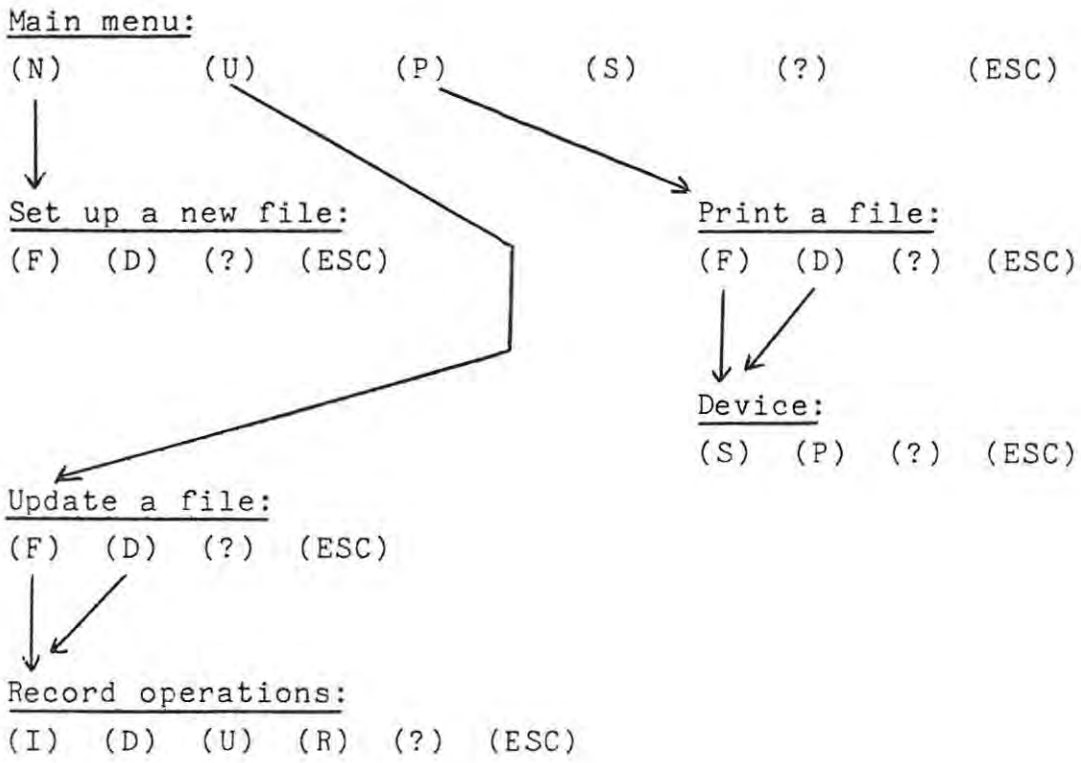


Figure 3.1
MICRO1 MENU HIERARCHY

Micro1 is menu driven and displays the active menu at the top of the screen. Figure 3.1 shows the hierarchical layout of the prompt menus.

The main menu is:
(N)ewfile (U)pdate (P)rint (S)etup (?)Help (ESC)Quit

The sub-sections that follow explain each of these options in more detail.

3.2.1 : (N)ewfile option.

On using the system for the first time the user has to set up new format and definition files.

(N)ewfile allows the user to set up a new format file and/ or a new definition file. These files are stored under the same workfile name with different extensions ie. 'workfile.FMT' and 'workfile.DEF' respectively.

SET UP A NEW FILE:
(F)ormat file (D)efinition file (?)Help (ESC)Quit

If the files already exist, they may be overwritten. (ESC)Quit gets you back to the 'main menu'.

Usually, the format file option would be chosen first. This option results in the following menu being printed on the screen.

EDITOR: INSERT ON
(<)Moveleft (>)Moveright (<-)Delete (Ins)Insert on/off (?)Help (ESC)Quit

The standard editing keys control cursor movement. The left arrow moves one character to the left and the right arrow one character to the right. The delete a character key deletes the character to the left of the cursor and the Ins key changes between insert and overstrike modes. (ESC)Quit gets you back to the 'set up a new file' menu.

These editing keys are active for all the input operations which follow. The first of these is a prompt for the user to enter the microinstruction width.

The width of the microinstruction is an integer value representing the number of bits that each microinstruction contains. Its value must be decided upon in the early stages of a new system development.

Once the width of the microinstruction has been entered, the user will be prompted to enter a numeric identifier (format number), description and format template definition for each unique microinstruction format. For example:

```
Format number: 1
Description:   2901 ALU
Format:        1x 2a 3x
```

The format number is a distinct integer valued identifier. Records in the format file are stored according to the format number and checks for duplicate records are made by comparing format numbers. The user is responsible for assigning a different format number to each distinct format. An error is reported if this is not done.

The description is a string expression. It should be chosen to assist the user in identifying the format at a later stage, eg. '2901 ALU'. This field is not used by the system and the user, although it is not

recommended, may leave it blank. It is present for the purpose of improving the readability of the format file and making later referencing more convenient.

The format definition is a template describing the function of the individual bits in the microinstruction. The letter 'x' represents a don't care field. Any other letter represents the 'active' part of the format. For example '1x 2a 3x' corresponds to 'xaaxxx'. The active field in this example comprises bits 2 and 3. Normally many micro-codes would use the same format template, as in the case of the different possible ALU functions.

The cycle of entering the format number, description and format definition is repeated until all format definitions have been recorded. When the prompt is given to enter the next format number, the <ESC> key may be typed to terminate the entries. If <ESC> is typed during a cycle, the current format definition record is discarded but previous ones are retained.

The next step is to enter the definition file records which is done by choosing the (D)efinition file option from the 'SET UP A NEW FILE' menu. The user will be prompted to enter a mnemonic identifier, value and format number for each distinct micro-order mnemonic definition. For example:

```
Mnemonic:       ALU AND
Value:          1010
Format number: 1
```

The mnemonic is a string expression which stores a name to be associated with a specific bit pattern and format. Each mnemonic must be unique as the records in the definition file are ordered according to this field. The mnemonic is therefore the name of a micro-order

which will be used later during the definition of the microinstructions.

The value field is a binary number representing the bit pattern with which the above mnemonic is to be associated, for example '1Ø1Ø'. This binary number will be substituted for the mnemonic when the microinstructions are assembled into binary fields for loading into the control store. 'N' or 'n' may be typed instead of a binary number, if the user wishes to insert the binary value into the microinstruction during the macro-instruction definition stage in the Micro2 program.

The format number is stored as an integer. It specifies the format with which the mnemonic is to be associated and therefore forms the connection between the records in the format and definition files. The binary value must be the same size as the active field in the given format. The system checks for this when the table file is being created.

The cycle of entering the mnemonic, value and format number is repeated until all the mnemonic definitions have been recorded. The <ESC> key may be typed when the next mnemonic is requested, to terminate the input session.

3.2.2 : (U)pdate option.

Once the format and definition file records have been entered they may be altered or corrected by selecting the (U)pdate option from the main menu.

(U)pdate allows the user to change the records in an existing format and/or definition file. One should not attempt to update records in an empty file or a file that does not exist.

UPDATE A FILE:
(F)ormat file (D)efinition file (?)Help (ESC)Quit

The available updating options are given on the lower level menu. (ESC)Quit gets you back to the 'main menu'.

Updating the format file puts the following list of allowed operations on the top of the screen.

RECORD OPERATIONS:
(I)nsert (D)elete (U)pdate (R)etrieve (?)Help (ESC)Quit

(I)nsert allows the user to insert a new format. Inserting a new format asks for the format number, a description and a format definition, in the same pattern as when setting up a new format file.
(D)elete allows the user to delete an existing format.
(U)pdate allows the user to change an existing format.
(R)etrieve allows the user to get a deleted record back.
(ESC)Quit gets you back to the 'update a file' menu.

If the (D)elete, (U)pdate or (R)etrieve options are selected, the system responds by asking for the appropriate format number.

The (U)pdate option presents the record information in the following way:

```
Format number: 1
Description:   2901 ALU
Format:        1x 2a 3x
```

One is only allowed to change the description and the format definition, using the editor provided by the system.

The updating session may be terminated by pressing the <ESC> key. If records have been deleted during the session, the system will ask the user if he is sure that he wishes to delete the marked records. If not the system will go through the records marked for deletion and attempt to retrieve them. If this however results in a duplicate record then that record is deleted anyway.

The updating of the records in the definition file is handled in very much the same way. The same record operations are allowed with the appropriate prompts worded differently. This is shown by the corresponding help message given below.

You are updating the definition file.
(I)nsert allows the user to insert a new mnemonic definition.
(D)elete allows the user to delete an existing mnemonic definition record.
(U)pdate allows the user to change an existing mnemonic definition record.
(R)etrieve allows the user to get a deleted record back.
(ESC)Quit gets you back to the 'update a file' menu.

3.2.3 : (P)rint option.

(P)rint, the third option in the main menu, enables the user to list the format and definition files.

The print menu is:
(F)ormat file (D)efinition file (?)Help (ESC)Quit

Both the format and the definition files may be output to either the screen or the printer as shown in the 'device' menu. (ESC)Quit gets you back to the 'main menu'.

DEVICE:
(S)creen (P)rinter (?)Help (ESC)Quit

If either of the files are listed to the screen, the records are printed out one at a time and the user has to press the space bar to get the next record. The <ESC> key may be typed to terminate the listing.

If either of the files are to be listed to the printer, the user is asked if the printer is ready. One must ensure that the printer is switched on and is connected before answering, to allow the listing of the whole file to proceed.

3.2.4 : (S)etup option.

The (S)etup option in the main menu sets in motion the procedures to construct the table file from the data in the format and definition files. The table file contains the target machine definition and is used as input to the Micro2 program.

The '(ESC)Quit' option may selected from the main menu to terminate the session and exit the Micro1 program. If the format or definition files have been changed, the system will ask if the user wishes to set up the table file. One may answer 'no' if for some reason one does not wish the (S)etup routine to be executed, for example, if the format and definition file information has not been completely entered or the table file has just been created.

3.2.5 : Error messages.

Below is a list of the error messages encountered during execution of the Micro1 program, with a brief explanation of each of them.

Error 1 : Integer value expected. The system expects the input to be an integer and must therefore contain no non-numeric characters.

Error 2 : Invalid format. Various checks are made to see if the formats entered are valid. The number of bits specified must tally with the width of the microinstruction and there must be at least one active field in the format.

Error 3 : Binary value expected. The system expects the number to be expressed in binary and it should therefore only contain 0's and 1's.

Error 4 : Duplicate format. Two records that have the same format number are not allowed.

Error 5 : Duplicate mnemonic. The micro-code mnemonics must be unique.

During the (S)etup option the following error messages may appear.
(1)Error with mnemonic : XXXX. Format number XX not found. Here the format to be associated with a mnemonic definition does not exist.
(2)Error with mnemonic : XXXX. The binary field of the mnemonic and the active part of the format are not the same length. This record is not included in the table file and the error should be rectified and the routines run again.
(3)The file(s), XXXX.FMT and or XXXX.DEF do not exist. Both of these files are necessary for this (the Setup) procedure. The format and definition files must be present in order for the table file to be created.

3.3 : Micro2 Program.

At the start of the Micro2 program session, the user is required to supply the name of the table file that was set up during the Micro1 program. The system assumes that the filename given has the extension '.TAB' and so only the file name stem has to be entered. The user may then select whether he wishes to include the help messages. The help messages are listed in appendix E.

Micro2, like Micro1, is menu driven and the active menu is displayed at the top of the screen.

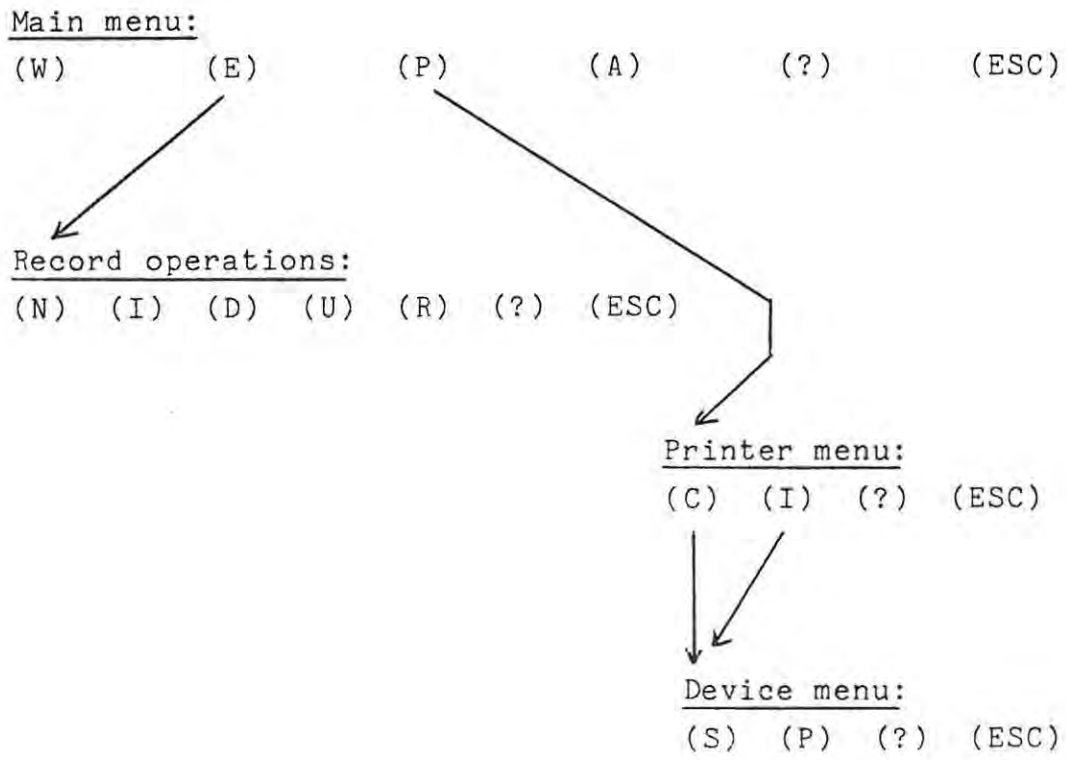


Figure 3.2
MICRO2 MENU HIERARCHY

The main menu is:
(W)orkfile (E)ditfile (P)rintfile (A)ssemble (?)Help (ESC)Quit

Figure 3.2 shows the hierarchical layout of the prompt menus. The sub-sections that follow explain each of these options in more detail.

3.3.1 : (W)orkfile option.

(W)orkfile sets up a working file on which all operations are performed. The macro-instruction definition files are given the extensions '.COD' and '.MIC', the instruction file is given the extension '.INS', the control store file '.BIN' and the decode file '.DEC'.

The editor menu is printed out along the top of the screen when the user is requested to enter the name of the workfile.

EDITOR: INSERT ON
(<)Moveleft (>)Moveright (<-)Delete (Ins)Insert on/off (?)Help (ESC)Quit

The standard editing keys control cursor movement and these editing keys are active for all the input operations that follow. The editor commands are the same as explained in section 3.2.1.

The workfile name is therefore the name of the file that contains the user's list of macro-instructions and their associated micro-codes. It is also the name given to all the files produced by the Micro2 program with each file having a different extension.

3.3.2 : (E)ditfile option.

(E)ditfile allows the user to set up a new macro-instruction definition file or to change an existing one. Inserting new records or changing existing records, proceeds one record at a time. Each record contains the macro-instruction mnemonic, a hexadecimal opcode and the corresponding microinstructions.

If the workfile has not been entered then it will be requested before record operations may proceed as given in the following menu.

RECORD OPERATIONS:
(N)ewfile (I)nsert (D)elete (U)pdate (R)etrieve (?)Help (ESC)Quit

The first operation that would have to be performed when using the program for the first time, would be to create a file using the (N)ewfile option. (N)ewfile allows the user to set up a new macro-instruction definition file. If the file already exists it may be overwritten. When a new file is specified the system reports that it already exists. This is because it has been assigned using the (W)orkfile option and it therefore has a valid entry in the directory.

The user is prompted to enter all the macro-codes in the form of a macro-instruction mnemonic, a hexadecimal opcode, followed by the appropriate microinstructions. For example:

```
Macro-instruction or micro-routine mnemonic: AND
Hexadecimal opcode: 2F
Micro-codes, one line per micro-instruction:
Areg #0000 & DZ & AND & RAMF & I-U & CY ;the first instruction
Breg #0001 & AB & OR & RAMA & I-M & CY;this is the second
; this is just a comment line
JUMPADDR #fetch & UNCOND & NOINCR ;jump to 'fetch' micro-routine
```

The macro-instruction is a mnemonic that is associated with a series of microinstructions. Each macro-instruction mnemonic must be unique. The micro-routine is a name associated with a sequence of microinstructions that do not have an associated opcode, for example, a 'fetch' routine.

The opcode is a two digit hexadecimal number representing an 8-bit macro-instruction in memory. Each opcode for a sequence of microinstructions must be unique. If this field is left empty, the system assumes that what follows is a micro-routine and not a macro-instruction definition. Both the macro-instruction mnemonic and the opcode fields are key fields for the file and both fields therefore have to be unique. One cannot have two mnemonics with the same opcode or two opcodes with the same mnemonic.

The micro-codes, also called a micro-routine or micro-code segment, are the microinstruction mnemonics associated with a certain macro-instruction. One line of micro-orders makes up one microinstruction. One or more microinstructions form a micro-code segment. The micro-orders are strung together with the sign '&'. Any binary fields that have to be included here should be preceded by a '#', for example:
LATER #1Ø1Ø1 where LATER was defined as a micro-order with the value of
'N' or 'n'.
A ';' denotes the start of a comment field.

The editor menu discussed previously is printed along the top of the screen and the associated functions apply. If the <ESC> key is typed at the end of a line then that line is ignored but previous lines are accepted. Typing the <RET> key at the end of a line accepts that line and automatically jumps to a new line. In this way all the microinstructions are accepted until an empty line is entered signaling the end of the micro-codes for that macro-instruction.

The system then requests the next macro-instruction mnemonic, opcode and associated micro-codes and in this way all the macro-instructions are entered. The input is terminated when the <ESC> key is pressed.

The rest of the record update operations are only meaningful after the (N)ewfile option has been used.

The (I)nsert command inserts a record into the existing macro-instruction definition file. This information is gathered in the same form as explained above, but this time only for one record ie. one macro-instruction mnemonic, its opcode and associated micro-codes.
The (D)elete option deletes a record from the workfile.
(U)pdate allows the user to update a record in the workfile.
(R)etrieve allows the user to get a deleted record back.
(ESC)Quit gets you back to the main menu.

The (D)elete, (R)etrieve and (U)pdate options act on one record at a time and they prompt the user to enter the desired macro-instruction mnemonic.

The (U)pdate option presents the record information in the format shown below.

```
Macro-instruction or micro-routine mnemonic: AND
Hexadecimal opcode: 2F
Micro-codes, one line per micro-instruction:
Areg #0000 & DZ & AND & RAMF & I-U & CY ;the first instruction
Breg #0001 & AB & OR & RAMA & I-M & CY;this is the second
;this is just a comment line
JUMPADDR #fetch & UNCOND & NOINCR ;jump to 'fetch' micro-routine
```

An enhanced editor is now used to update the microinstruction in this micro-code segment. The menu is given below.

EDITOR: INSERT ON
(<)Moveleft (>)Moveright (<-)Delete (Ins)Insert on/off (?)Help (ESC)Quit
(UP ARROW)Insert line (DOWN ARROW)Delete line

The Moveleft, Moveright, Delete and Insert functions operate in the manner previously described. The Insert line and Delete line functions have been added. The (UP ARROW) key inserts a line above the present cursor line. The (DOWN ARROW) key deletes the line that the cursor is on. Lines are automatically inserted if one is at the end of the text. The <RET> key must be typed at the end of each line to accept the changes. Pressing the <ESC> key will terminate the update session.

3.3.3 : (P)rint option.

(P)rintfile allows the user to list the Micro2 files.

The print menu is:
(M)acro-instruction (I)nstruction file (D)ecode file (C)ontrol store
(?)Help (ESC)Quit

(M)acro-instruction lists the user's workfile of macro-instruction definitions. The records may be listed to either the screen or the printer.
(I)nstruction file, lists the hexadecimal file of microcodes. This information is the same as is contained in the control store file but it is presented in a different format, ie. hexadecimal instead of binary.
(D)ecode file, lists the look-up table of the opcodes and their corresponding addresses.
(C)ontrol store, lists the binary control store file. This file can be listed on the screen but is difficult to read and should therefore

preferably be listed to the printer.
(ESC)Quit, gets you back to the main menu.

Once a file has been selected for printing, the following 'device' menu is listed. Files may be listed to either the screen or the printer.

DEVICE:
(S)creen (P)rinter (?)Help (ESC)Quit

If the files are listed to the screen, the records are presented one at a time and the user has to press the space bar to list the next record. If the <ESC> key is typed, the listing is terminated.

If the files are listed to the printer, the user is asked to ensure that the printer is ready before proceeding with the operation.

3.3.4 : (A)ssemble option.

The forth option in the main menu is the (A)ssemble option, which sets in motion the procedures to produce the control store file from the file of macro-instruction definitions.

The "don't care" fields in the microinstruction are those that are not used for that specific micro-operation. Not all the fields are used in every microinstruction and the user may select whether these are to be filled with Ø's or 1's.

The system informs the user that he may terminate the assembly process by holding the space bar down.

Once the binary or control store file has been satisfactorily produced, the user may select the (ESC)Quit option from the main menu to exit the

Micro2 program.

3.3.5 : Error messages.

The following error messages may appear during the use of the Micro2 program.

Error 1 : Overlapping fields. This means that two micro-orders in that microinstruction are competing for the same bit position or positions. This is obviously an undesirable state.

Error 2 : u-Instruction not found. Here the micro-order mnemonic has not been defined and can therefore not be found in the table file. Please note that the system as defined is case sensitive and will therefore distinguish between the instructions, 'add' and 'ADD'.

Error 3 : Duplicate macro-instruction or duplicate opcode. These are both key fields in the macro-instruction definition file and duplicate records are therefore not allowed.

Error 4 : Invalid hexadecimal number. The system expects the number given to be a two digit hexadecimal number. The letters 'A' to 'F' should be entered as capital letters.

Error 5 : Undefined label or microroutine. A jump within the microcode can only be made to a defined label or microroutine.

Error 6 : Duplicate label definition. Jump labels within the microcode must be unique.

3.4 : Emulator Program.

After the emulator program session has been initiated, the user is requested to select whether he wishes to include the help messages. This help facility operates in the same way as explained for the Micro1 program. The user is requested to input the name of the control store file, the one set up by the Micro2 program, and has to decide whether or not to simulate the action of the ALU hardware. Finally the user is prompted to enter the name of the file that contains the high-level program to be run. This macroprogram is then loaded into the main memory.

The Emulator is menu driven and the active menu is displayed on the top of the screen. Figure 3.3 shows the hierarchical layout of the prompt menus in this program.

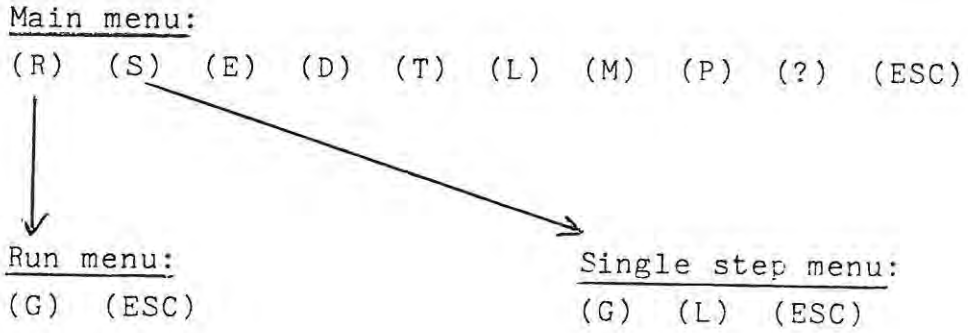


Figure 3.3
EMULATOR MENU HIERARCHY

The main menu options are given below.

```
MAIN MENU:                                                    TRACE = FALSE
(R)un   (S)ingle   (E)dit   (D)-bus   (T)race   (L)ook   (M)PC   (P)eek
(?)Help   (ESC)Quit
```

The sub-sections that follow explain each of these options in more detail.

3.4.1 : (R)un option.

(R)un executes the program in pseudo real time. The ALU hardware is run very slowly so at to keep in step with the Emulator software. Once the (G)o option has been selected, the screen is cleared and the user is presented with the normal view of the machine. I/O is now needed to monitor the execution. The user should ensure that the trace mode is inactive before using this option, as this prevents the debugging messages from being listed on the screen.

The microinstructions are executed as quickly as possible and the user may halt the execution by holding the space bar down.

3.4.2 : (S)ingle option.

(S)ingle allows the user to single step through the program. The microinstructions are executed one at a time after each (G)o command is given. Each microinstruction that is being executed is listed to the screen. Further instructions within this level are outlined below.

(G)o executes one microinstruction.

(L)ook allows the user to look at the specific register values. This is discussed further in section 3.4.6.
(ESC)Quit terminates the execution and returns control to the main menu.

3.4.3 : (E)dit option.

(E)dit enables the user to alter the microinstructions in the control store file.

The user is requested to enter the address of the microinstruction, in the control store file, that is to be updated.

```
EDITOR:                                                    INSERT ON
(<)Moveleft  (>)Moveright  (<-)Delete  (Ins)Insert on/off  (?)Help
(ESC)Quit
```

The above editor menu is printed out along the top of the screen for the microinstruction address request and the subsequent updating. The editor commands are the same as explained in section 3.2.1. The <RET> key must be typed to accept the changes. The user should note that changes made here will dissapear if the microprogram is re-assembled from the Micro2 program. Permanent updates would have to be made to the microinstruction definitions.

These editing keys are active for all the input operations that follow.

3.4.4 : (D)-bus option.

The (D)-bus command is used to change the D-bus value. The old value is printed out and the user is requested to enter the new value. The D-bus

value is the buffer that is used as input to the ALU.

3.4.5 : (T)race option.

The (T)race command alters the trace mode in a toggle fashion. The status of the trace mode is given in the top right hand corner of the screen. When the trace mode is active, the values of various registers are printed out after the execution of each microinstruction.

3.4.6 : (L)ook option.

The (L)ook option allows the user to view specific register values within the ALU. This option is specific to the type of ALU hardware that is used with the system. For the present Am2901 based ALU, the 16 working registers are listed to the screen.

3.4.7 : (M)PC option.

The (M)PC command is invoked to change the MPC value. The old value of the MPC is printed out and the user is requested to enter the new value. The MPC is the Micro-Program Counter, that is, the control store program counter.

3.4.8 : (P)eek option.

The (P)eek command affords the user a view of the main memory. To examine the contents of a memory location, the address is entered and the system responds by printing the contents of main memory at that address. The memory location value cannot be changed with this command.

The <ESC> key may be typed from the main menu to terminate the session and exit the Emulator program.

3.4.9 : Error messages.

The only error message here, is one to inform the user that an integer value was expected, and that the input entered contained non-numeric characters. The system also issues a warning if the user is attempting to access a control store address that does not contain a valid microinstruction.

3.5 : AssemMW Program.

After the AssemMW program has been initiated, the user is requested to enter the name of the control store file. This is the file that was set up by the Micro2 program. Thereafter the name of the macroprogram file has to be entered. The system assumes that the file containing the macro-instruction mnemonics has already been input and exists in a file with the extension '.SCE'.

AssemMW is menu driven and the active menu is displayed at the top of the screen.

The main menu is:
(E)dit (A)ssemble (P)rint (?)Help (ESC)Quit

3.5.1 : (E)dit option.

The (E)dit option enables the user to change the value at a chosen memory location in the object file. The user is prompted to enter the address of the location that is to be changed and that address is then displayed along with the old value. This value may be changed by simply entering the new value as shown below:

```
5  16  _ (type new value)
```

3.5.2 : (A)ssemble option.

The (A)ssemble command sets in motion a two pass assembler that converts the source file, XXXX.SCE, into the object file, XXXX.OBJ. It is the object file that contains the actual hexadecimal opcodes and operands, and is used as input to the Emulator program.

3.5.3 : (P)rint option.

(P)rint allows the user to list the object file to either the screen or the printer.

The <ESC> key may be typed from the main menu to exit the AssemMW program.

BIBLIOGRAPHY.

[ADV] Advanced Micro Devices : The Am2900 Family Data Book with Related Support Chips; AMD, (1979).

[AGR] Agrawala Ashok K. , Rauscher Tomlinson G. : Foundations of Microprogramming; Academic Press, (1976).

[AME] Amerson Frederic C. : Simplicity in a Microcoded Computer Architecture; Hewlett-Packard Journal, September, (1985), pp.7-12.

[BAB] Baba T. , Hagiwara H. : The MPG System: A Machine-Independent Efficient Microprogram Generator; IEEE Transactions on Computers, Volume C-30, Number 6, (1981), pp.373-395.

[BAL1] Balakrishnan M. , Madan B.B. , Bhatt P.C.P. : A Survey of Microprogramming Languages; Microprocessing and Microprogramming, Volume 17, (1986), pp.19-28.

[BAL2] Ballieu G. , Lewi J. , Williams Y.D. : A Microprogramming Language at Register Transfer Level; Microprocessing and Microprogramming, Volume 8, (1981), pp.179-188.

[BEH] Behr P. , Goli W.K. , Gueth R. : Education in Firmware Engineering and Microprogramming; Microprocessing and Microprogramming, Volume 8, (1981), pp.153-166.

[CHA] Charlton C.C. , Elliot D. , Leng P.H. : An interactive Software System for Microcode Development; Microprocessing and Microprogramming, Volume 13, (1984), pp.105-114.

[CHR] Chroust Gerhard : Guest Editors Preface; Microprocessing and Microprogramming, Volume 8, (1981), pp.137-140.

[CLA1] Clayton Peter G. : Microprogramming and Bit-Slice Logic; Department of Computer Science, Rhodes University, (1984).

[CLA2] Clayton Peter G. : A Code Generator Synthesiser for the Non-Specialist; Software - Practice and Experience, Volume 16(8), (1986), pp.751-760.

[CLA3] Clayton Peter G. : Hands-On Microprogramming for Computer Science Students; Department of Computer Science, Rhodes University, (1986).

[COL] Colard D. : Development Aid Programs for Bit-Slice Based Systems; Microprocessing and Microprogramming, Volume 7, (1981), pp.58-65.

[COR] Corcoran Peter : The SUMA Microprogramming System; Microprocessing and Microprogramming, Volume 7, (1981), pp.37-45.

[DAT] Dataweek : Meta Step Assembler Speeds Bit-Slice Coding; Dataweek, Volume 9, No 8, April, (1986).

[DAV] Davies A.C. , Ibrahim D. : A Basis for Laboratory work with Bit - Slice Micoprogrammable Microprocessors; The Challenge of Microprocessors, Editors : Hartley Michael J. and Buckley Anne; Manchester United Press, (1979).

[DUE] Duetsch L.P. : Experience with a Microprogrammed Interlisp System; Xerox Corporation, CA, (1978), pp.128-129.

[DIM] Dimond K.R. , King J.A. : A Flexible Development System for Microprogrammable Microprocessors; The Challenge of Microprocessors, Editors : Hartley Michael J. and Buckley Anne; Manchester United Press, (1979).

[EDE] Edel W. : The 8002 Universal Microprocessing Development Aid System; Euromicro, Volume 5, (1979), pp.25-30.

[EGG] Eggebrecht Lewis C. : Interfacing to the IBM Personal Computer; Howard W. Sams, (1983).

[FLE] Fletcher William I. : An Engineering Approach to Digital Design; Prentice-Hall, (1980).

[GIB] Gibson Ray : Microprogramming and Microprocessors: Investigation of Development Systems; The Challenge of Microprocessors, Editors : Hartley Michael J. and Buckley Anne; Manchester United Press, (1979).

[GOL] Goldsbrough Paul F. : Microcomputer Interfacing with the 8255 PPI Chip; Howard W. Sams, (1979).

[GRA] Grant Richard P.J.S. : Microcomputer Interfacing; Department of Physics and Electronics, Rhodes University, (1986).

[GRI] Griss M.L. , Swanson M.R. : A Microprogrammed LISP Machine for the Burroughs B1726; Computer Science Department, University of Utah, Utah, (1978), pp.15-25.

[GRO] Grossman B. , Kwee E. , Lehmann A. : Practical Experiences with Vertical Migration; Microprocessing and Microprogramming, Volume 12, (1983), pp.185-192.

[KER] Kerner Helmut : Microprogramming by Data Flow; Microprocessing and Microprogramming, Volume 12, (1983), pp.181-184.

[IBM] IBM Personal Computer Technical Reference; International Business Machines Corporation, (1981).

[KRA] Kraft George D. , Toy Wing N. : Microprogrammed Control and Reliable Design of Small Computers; Prentice-Hall, (1981).

[KRU] Krutz Ronald L. : Microprocessors and Logic Design; John Wiley, (1980).

[LIE] Liebig Hans : Microprogramming with Microprocessors; Microprocessing and Microprogramming, Volume 12, (1983), pp.43-52.

[LEW] Lewis T.G. , Schriver B.D. : Special Issue on Microprogramming Tools and Techniques: Introduction; IEEE Transactions on Computers, Volume C-30, Number 7, (1981), pp.257-259.

[LUQ] Luque E. , Ripoll A. : Microprogramming:A Tool for Vertical Migration; Microprocessing and Microprogramming, Volume 8, (1981), pp.219-228.

[MCG] McGlynn Daniel R. : Modern Microprocessor System Design; John Wiley, (1980).

[MEA] Mead Carver, Conway Lynn : Introduction to VLSI Systems; Addison-Wesley, (1980).

[MEI] Meith W.H. , Richter L. : MMDS - A Microprogram Development Tool; Euromicro, (1981), pp.261-268.

[MEZ1] Mezzalama M. , Prinetto P. , Visintin I. : A Hierarchical Integrated System for Microcode Development; Euromicro, (1981), pp.251-258.

[MEZ2] Mezzalama M. , Prinetto P. : A Strategy for Simulating Bit-Slice Based Microprogrammable Systems; Microprocessing and Microprogramming,

Volume 7, (1981), pp.334-343.

[MEZ3] Mezzalama Marco , Prinetto Paolo : Microprogram Simulation Using a Structured Microcode Model; Microprocessing and Microprogramming, Volume 13, (1984), pp.299-314.

[MIC] Mick John , Brick James : Bit-Slice Microprocessor Design; McGraw-Hill, (1980).

[MON] Monchaud S. , Prat R. : A Low-Cost Microprogram Development System Loader and Tester; Euromicro, Volume 5, (1979), pp.225-234.

[MOR] Morgan G. , Pack R. , Lala P.K. : An Educational Bit-Slice Micrprogrammable Tutor; Department of Computer Science, University of York, YCS.69, (1984).

[MUE] Mueller Robert A. , Varghese Joseph : Applying Algebraic Simulation to Machine-Independent Microcode Synthesis; Microprocessing and Microprogramming, Volume 11, (1983), pp.107-115.

[MYE1] Myers Glenford J. : Digital System Design with LSI Bit-Slice Logic; John Wiley, (1980).

[MYE2] Myers Glenford J. , Hocker David G. : The Use of Software Simulators in the Testing and Debugging of Microprogram Logic; IEEE Transactions on Computers, Volume C-30, Number 7, (1981), pp.519-523.

[OBR] Obrebska Monika : Efficiency and Performance Comparison Of Different Design Methodologies for Control Part of Microprocessors; Microprocessing and Microprogramming, Volume 10, (1982), pp.163-178.

[OPL] Opler Asher : Fourth Generation Software; Microprocessing and Microprogramming, Volume 8, (1981), pp.146-148.

[PUT] von Puttkamer E. : A Microprogrammed Lisp Machine; Microprocessing and Microprogramming, Volume 12, (1983), pp.9-14.

[RAU] Rauscher T.G. , Adams P.M. : Microprogramming: A Tutorial and Survey of Recent Developments; IEEE Transactions on Computers, Volume C-29, Number 1, (1980), pp.2-18.

[REN] Renyi I. , Lovaszi M. : Parallel Picture Processing Using Microprogrammable Bit-Slice Microprocessors; Microprocessing and Microprogramming, Volume 9, (1982), pp.67-75.

[SCH] Schreiner-Novick N.A. : 15th Workshop on Microprogramming; Microprocessing and Micrprogramming, Volume 11, (1983), pp.141-162.

[SKO] Skordalakis E. : Towards a More Flexible Microlanguage for Bit-Sliced Microcomputers; Microprocessing and Microprogramming, Volume 7, (1981), pp.46-57.

[SOM] Sommerville J. F. : Towards Machine-Independent Microprogramming; Euromicro, Volume 5, (1979), pp.219-224.

[SRI] Sridar R. , Manwaring Mark L. : An Automatic Microcode Generator for High Level Language Machines; Microprocessing and Microprogramming, Volume 18, (1986), pp.263-268.

[STA] Stankovic John A. : Improving System Structure and its Affect on Vertical Migration; Microprocessing and Microprogramming, Volume 8, (1981), pp.203-218.

[TEX] Texas Instruments : The TMS 7000 family; Texas Instruments, (1985).

[TSU] Tsuchiya M. : FREM:Firmware Requirements Engineering Methodology; Microprocessing and Microprogramming, Volume 8, (1981), pp.167-178.

[TUR] TURBO Pascal version 3.0 Reference Manual; Borland International, (1985).

[VAX] VAX Hardware Handbook; Digital Equipment Corporation, (1980).

[WHI] White Donnamaie E. : Bit-Slice Design : Controllers and ALU's; Garland STPM, (1981).

[WIL] Wilkes Maurice : The Best Way to Design an Automatic Calculating Machine; Microprocessing and Microprogramming, Volume 8, (1981), pp.141-144.

[WIN] Winkel David, Prosser Franklin : The Art of Digital Design; Prentice-Hall, (1980).

[ZIN] Zincke G.D. : Why is Microprogramming Difficult? Some thoughts about a Generally Accepted Problem; Microprocessing and Microprogramming, Volume 8, (1981), pp.149-152.

APPENDICES

Appendices A to D.

Program listings may be obtained from:

The Secretary
Department of Computer Science
Rhodes University
Grahamstown
6140
South Africa

Phone 0461 - 22023

LISTING OF THE HELPFILE FOR THE MICRO1 PROGRAM

```
#1
HELP 1

(N)ewfile allows you to set
up a new format file and/or
a new definition file.

(U)pdate allows you to update
the records in an existing
format and/or definition file.

(P)rint allows you to list
the format and definition
files.

(S)etup creates the table
file from the data in the
format and definition files.

(ESC)Quit, to end the session.
*

#2
HELP 2

(F)ormat file allows you to
set up a new format file.

(D)efinition file allows you
to set up a new definition
file.

(ESC)Quit gets you back to
the 'main menu'.
*

#3
HELP 3

The width of the micro-
instruction is an integer
value representing the number
of bits that each micro-
instruction contains.

(ESC)Quit gets you back to
the 'set up a new file' menu.
*

#4
HELP 4

Format number is an integer
```

```
value. Each format must have
a different distinct format
number.
*

#5
HELP 5

Description is a string
expression. It should help
the user to identify the
format at a later stage.

eg. '2901 ALU'.
*

#6
HELP 6

Format is the actual format
template. The letter 'x'
represents a don't care field.
Any other letter represents
the 'active' part of the
format.

eg. '1x 2a 3x'  ==>  xaaxxx
   the active field being
   bits 2 and 3.
*

#7
HELP 7

Mnemonic is a string
expression. Each mnemonic must
be unique. It is associated
with a specific bit pattern
and format.

eg. 'ALU AND'.
*

#8
HELP 8

Value is a binary field. It is
the bit pattern with which the
above mnemonic is to be
associated.

eg. '1010'.

'N' or 'n' may be typed here
instead, if the user wishes
```

```
to insert the actual value at
the macro-instruction
definition stage later.
*

#9
HELP 9

Format number is an integer.
It specifies the format with
which the mnemonic is to be
associated.
*

#10
HELP 10

(F)ormat file allows you to
update the format file.

(D)efinition file allows you
to update the definition file.

(ESC)Quit gets you back to
the 'main menu'.
*

#11
HELP 11

You are updating the format
file.

(I)nsert allows you to insert
a new format.

(D)elete allows you to delete
an existing format.

(U)pdate allows you to change
an existing format.

(R)etrieve allows you to get
a deleted record back.

(ESC)Quit gets you back to the
 'update a file' menu.
*

#12
HELP 12

You are updating the
definition file.
```

```
(I)nsert allows you to insert
a new mnemonic.

(D)elete allows you to delete
an existing mnemonic.

(U)pdate allows you to change
an existing mnemonic.

(R)etrieve allows you to get
a deleted record back.

(ESC)Quit gets you back to the
 'update a file' menu.
*

#13
HELP 13

(F)ormat file allows you to
list the format file.

(D)efinition file allows you
to list the definition file.

(ESC)Quit gets you back to
the 'main menu'.
*

#14
HELP 14

(S)creen lists the file on
the screen.

(P)rint lists the file on the
printer.

(ESC)Quit terminates the
screen listing.
*

#15
HELP 15

Press the space bar to
continue the listing one
record at a time.

(ESC) terminates the listing.
*
```

LISTING OF THE HELPFILE FOR THE MICRO2 PROGRAM

```
#1
HELP 1

(W)orkfile sets up a workfile
on which all operations are
performed.

(E)ditfile allows you to set
up a new codefile or to
change an existing one.

(P)rintfile allows you to list
the workfile or the control
store file.

(A)ssemble creates the binary
control store file.

(ESC)Quit, to end the session.
*

#2
HELP 2

(N)ewfile allows you to set
up a new codefile.

(I)nsert, inserts a record
into the existing workfile.

(D)elete, deletes a record
from the workfile.

(U)pdate allows you to update
a record in the workfile.

(R)etrieve allows you to get
a deleted record back.

(ESC)Quit, gets you back to
the main menu.
*

#3
HELP 3

(M)acro-instruction lists the
user's workfile of macro-
instruction definitions.

(I)nstruction file, lists the
hexadecimal file of
microcodes.
```

(D)ecode file, lists the
look-up table of the opcodes
and their corresponding
addresses.

(C)ontrol store, lists the
binary control store file.

(ESC)Quit, gets you back to
the main menu.
*

#4
HELP 4

(S)creen lists the file to
the screen.

(P)rinter lists the file to
the printer.

(ESC)Quit gets you back to
the printer menu.
*

#5
HELP 5

The workfile name is the name
of the file that contains the
user's list of macro-
instructions and their
associated micro-codes.
*

#6
HELP 6

The macro-instruction is a
mnemonic that is associated
with a series of micro-
instructions. Each macro-
instruction mnemonic must
be unique.
The micro-routine is a name
associated with a sequence of
microinstructions that do not
have an associated opcode
eg. the 'fetch' routine.
*

#7
HELP 7

```
The micro-codes are the
microinstruction mnemonics
associated with a certain
macro-instruction. One line
of micro-orders makes up one
microinstruction. One or
more microinstructions form
a section of micro-code.
The micro-orders are strung
together with the sign '&'.
Any binary fields that have
to be included here should be
preceded by a '#'.
eg 'LATER #10101' where LATER
was defined as a micro-order
with the value of 'N' or 'n'.
A ';' denotes the start of a
comment field.
*

#8
HELP 8

Press the space bar to
continue the listing one
record at a time.

(ESC)Quit terminates the
listing.
*

#9
HELP 9

The opcode is a two digit
hexadecimal number
representing an 8-bit opcode
in memory. Each opcode for a
sequence of microinstructions
must be unique.
If an empty field is entered
here the system assumes that
this is a micro-routine and
not a macro-instruction.
*
```

LISTING OF THE HELPFILE FOR THE EMULATOR PROGRAM

```
#1
HELP 1

(R)un allows you to run the
program in pseudo real time.

(S)ingle step thru program.

(E)dit a microinstruction.

(D)-bus to change the D-bus.

(T)race mode toggle on/off.

(L)ook to view registers.

(M)PC to change the MPC.

(P)eek at memory locations.

(ESC)Quit to terminate.
*

#2
HELP 2

(G)o starts the execution.

Hold the space bar down to
stop execution.
*

#3
HELP 3

(G)o executes one
microinstruction.

(L)ook allows you to look at
specific register values.

(ESC)Quit to terminate the
execution.
*

#4
HELP 4

The editor commands are as
shown at the top of the
screen. Press <RET> to
accept the changes.
*
```

```
#5
HELP 5

This is an address in the
Control Store of the micro-
instruction that is to be
updated.
*

#6
HELP 6

This is the value that is
input to the ALU.
*

#7
HELP 7

The MPC is the Micro-Program
Counter, ie. the Control
Store program counter.
*
```

Appendix F : Micro-order function tables

The present version of the Microprogram Control Unit requires the first 24 bits of the microinstruction to be reserved for its functions. These functions have been designed with the Triple-M machine in mind but were based on those provided by Advanced Micro Devices' Am2909 and Am2910 controller/sequencers.

A field to hold a jump address or a constant value:

I1 to I12	Action	Mnemonic
Binary address	Specified by I14 and I15	None

Increment the Micro-Program Counter:

I13	Action	Mnemonic
0	Increment	INCR
1	No increment	NOINCR

Branching:

I14 I15	Action	Mnemonic
00	No Branch	NOBRN
01	Unconditional	UNCOND
10	Conditional	COND
11	Constant field	CONST

Memory access:

I16 I17 I18	Action	Mnemonic
000	No memory access	NMA
001	Data register -> D-buffer	DR-DB
010	Y-buffer -> Address register	YB-AR

011	Y-buffer -> Data register	YB-DR
100	Data register -> Micro Latch	DR-UIR
101	Micro Latch -> UIR	MPC-AR
110	Data register -> MIR and decode	DR-MIR
111	Data register -> Address register	DR-AR

Clock field:

I19	Action	Mnemonic
0	Don't send a clock pulse	CN
1	Send a clock pulse	CY

Free:

I20 to I24	Action	Mnemonic
Not used	None	None

Advanced Micro Device's Am2904 status and shift control unit functions are tabulated below [ADV].

Micro status register instruction codes:

I25 to I30	Action	Mnemonic
000000	MSR -> uSR	MX-UX
000001	Set uSR	1-UX
000010	Register swap	MX+UX
000011	Reset uSR	0-UX
000100	Load uSR from I's	I-U
000111	Load with overflow retain	I-UO
001000	Reset zero bit	0-UZ
001001	Set zero bit	1-UZ

001010	Reset carry bit	0-UC
001011	Set zero bit	1-UC
001100	Reset sign bit	0-UN
001101	Set sign bit	1-UN
001110	Reset overflow bit	0-UOVR
001111	Set overflow bit	1-UOVR
011000	Load with carry invert	I-UO

Machine status register instruction codes:

I25 to I30	Action	Mnemonic
000000	Load MSR from Y	YX-MX
000001	Set MSR	1-MX
000010	Register swap	UX-MX
000011	Reset uSR	0-MX
000100	Swap Mc and Movr	I-MO
000101	Invert MSR	MX-MX
001000	Load with carry invert	I-MC
001111	Load directly from I's	I-M

Carry-in control multiplexor instruction codes:

I26 to I30,I36,I37	Action	Mnemonic
0000000	Reset carry-in	CO=0
0000010	Set carry-in	CO=1
0000001	Load Cx into carry-in	CO=CX
0000011	Load uC into carry-in	CO=UC
0010011	Load not uC	CO=NUC
0000111	Load MC into carry-in	CO=MC
0010111	Load not MC	CO=NMC

Shift linkage multiplexer instruction codes:
(Refer to the text for these seldom used instructions.)

Condition code output (CT) instruction codes:

I25 to I30	Action	Mnemonic
000010	(uN+uO) + uZ	u0
100010	(uN.uO) . (not uZ)	u1
010010	uN + uO	u2
110010	uN . uO	u3
001010	uZ	u4
101010	not uZ	u5
011010	uO	u6
111010	not uO	u7
000110	uC + uZ	u8
100110	uC . uZ	u9
010110	uC	uA
110110	not uC	uB
001110	(not uC) + uZ	uC
101110	uC . (not uZ)	uD
011110	uN	uE
111110	not uN	uF
000001	(Mn+Mo) + Mz	M0
100001	(Mn.Mo) . (not Mz)	M1
010001	Mn + Mo	M2
110001	Mn . Mo	M3
001001	Mz	M4
101001	not Mz	M5
011001	Mo	M6
111001	not Mo	M7
000101	Mc + Mz	M8
100101	Mc . Mz	M9

010101	Mc	MA
110101	not Mc	MB
001101	(not Mc) + Mz	MC
101101	Mc . (not Mz)	MD
011101	Mn	ME
111101	not Mn	MF

(The direct checks on the 'I' lines have not been included.)

Advanced Micro Device's Am2901 arithmetic and logic unit functions are given below [ADV].

ALU source operand control:

I40 to I42	Action	Mnemonic
000	R=A, S=Q	AQ
100	R=A, S=B	AB
010	R=0, S=Q	ZQ
110	R=0, S=B	ZB
001	R=0, S=A	ZA
101	R=D, S=A	DA
011	R=D, S=Q	DQ
111	R=D, S=0	DZ

ALU function control:

I43 to I45	Action	Mnemonic
000	R plus S	ADD
100	S minus R	SUBR
010	R minus S	SUBS
110	R or S	OR
001	R and S	AND

101	(not R) and S	NOTRS
Ø11	R EX-OR S	EXOR
111	R EX-NOR S	EXNOR

ALU destination control:

I46 to I48	Action	Mnemonic
ØØØ	F->Q, Y=F	QREG
1ØØ	Y=F	NOP
Ø1Ø	F->B, Y=A	RAMA
11Ø	F->B, Y=F	RAMF
ØØ1	F/2->B, Q/2->Q, Y=F	RAMQD
101	F/2->B, Y=F	RAMD
Ø11	2F->B, 2Q->Q, Y=F	RAMQU
111	2F->B, Y=F	RAMU

File : proto1

LISTING OF THE FORMAT FILE
==========

```
Format number: 1
Description:   2904-micro-stat
Format:        24x 6a 7x 2a 17x

Format number: 2
Description:   2904-macro-stat
Format:        24x 6a 7x 2a 17x

Format number: 3
Description:   2904-carry-in
Format:        25x 5a 5x 2a 19x

Format number: 4
Description:   2904-shift-link
Format:        30x 5a 21x

Format number: 5
Description:   2901-alu-source
Format:        39x 3a 14x

Format number: 6
Description:   2901-alu-function
Format:        42x 3a 11x

Format number: 7
Description:   2901-alu-destination
Format:        45x 3a 8x

Format number: 8
Description:   2901-Areg
Format:        48x 4a 4x

Format number: 9
Description:   2901-Breg
Format:        52x 4a

Format number: 10
Description:   2904-cond-code
Format:        24x 6a 26x
```

```
Format number: 11
Description:   2904-s-carry-in
Format:        35x 2a 19x

Format number: 12
Description:   ward2909-jump
Format:        12a 44x

Format number: 13
Description:   ward2909-incr
Format:        12x 1a 43x

Format number: 14
Description:   ward2909-branch
Format:        13x 2a 41x

Format number: 15
Description:   ward2909-access
Format:        15x 3a 38x

Format number: 16
Description:   ward2909-clock
Format:        18x 1a 37x

Format number: 17
Description:   not CEu and CEm
Format:        37x 2a 17x
```

File : proto1

LISTING OF THE MNEMONIC FILE
==========

Mnemonic: AQ
Value: 000
Format no.: 5

Mnemonic: AB
Value: 100
Format no.: 5

Mnemonic: ZQ
Value: 010
Format no.: 5

Mnemonic: ZB
Value: 110
Format no.: 5

Mnemonic: ZA
Value: 001
Format no.: 5

Mnemonic: DA
Value: 101
Format no.: 5

Mnemonic: DQ
Value: 011
Format no.: 5

Mnemonic: DZ
Value: 111
Format no.: 5

Mnemonic: ADD
Value: 000
Format no.: 6

Mnemonic: SUBR
Value: 100
Format no.: 6

```
Mnemonic:   SUBS
Value:      010
Format no.: 6

Mnemonic:   OR
Value:      110
Format no.: 6

Mnemonic:   AND
Value:      001
Format no.: 6

Mnemonic:   NOTRS
Value:      101
Format no.: 6

Mnemonic:   EXOR
Value:      011
Format no.: 6

Mnemonic:   EXNOR
Value:      111
Format no.: 6

Mnemonic:   QREG
Value:      000
Format no.: 7

Mnemonic:   NOP
Value:      100
Format no.: 7

Mnemonic:   RAMA
Value:      010
Format no.: 7

Mnemonic:   RAMF
Value:      110
Format no.: 7

Mnemonic:   RAMQD
Value:      001
Format no.: 7
```

```
Mnemonic:   RAMD
Value:      101
Format no.: 7

Mnemonic:   RAMQU
Value:      011
Format no.: 7

Mnemonic:   RAMU
Value:      111
Format no.: 7

Mnemonic:   Areg
Value:      n
Format no.: 8

Mnemonic:   Breg
Value:      n
Format no.: 9

Mnemonic:   0-UZ
Value:      00010001
Format no.: 1

Mnemonic:   1-UZ
Value:      10010001
Format no.: 1

Mnemonic:   0-UC
Value:      01010001
Format no.: 1

Mnemonic:   1-UC
Value:      11010001
Format no.: 1

Mnemonic:   0-UN
Value:      00110001
Format no.: 1

Mnemonic:   1-UN
Value:      10110001
Format no.: 1
```

```
Mnemonic:   Ø-UOVR
Value:      01110001
Format no.: 1

Mnemonic:   1-UOVR
Value:      11110001
Format no.: 1

Mnemonic:   MX-UX
Value:      00000001
Format no.: 1

Mnemonic:   1-UX
Value:      10000001
Format no.: 1

Mnemonic:   MX+UX
Value:      01000001
Format no.: 1

Mnemonic:   Ø-UX
Value:      11000001
Format no.: 1

Mnemonic:   I-UO
Value:      11100001
Format no.: 1

Mnemonic:   I-UC
Value:      00011001
Format no.: 1

Mnemonic:   I-U
Value:      00100001
Format no.: 1

Mnemonic:   YX-MX
Value:      00000010
Format no.: 2

Mnemonic:   1-MX
Value:      10000010
Format no.: 2
```

```
Mnemonic:   UX-MX
Value:      01000010
Format no.: 2

Mnemonic:   Ø-MX
Value:      11000010
Format no.: 2

Mnemonic:   MX-MX
Value:      10100010
Format no.: 2

Mnemonic:   I-MO
Value:      00100010
Format no.: 2

Mnemonic:   I-MC
Value:      00010010
Format no.: 2

Mnemonic:   I-M
Value:      11110010
Format no.: 2

Mnemonic:   CO=Ø
Value:      ØØ
Format no.: 11

Mnemonic:   CO=1
Value:      1Ø
Format no.: 11

Mnemonic:   CO=CX
Value:      Ø1
Format no.: 11

Mnemonic:   CO=UC
Value:      0000011
Format no.: 3

Mnemonic:   CO=NUC
Value:      0010011
Format no.: 3
```

Mnemonic: CO=MC
Value: 0000111
Format no.: 3

Mnemonic: CO=NMC
Value: 0010111
Format no.: 3

Mnemonic: Z0Z0
Value: 00000
Format no.: 4

Mnemonic: Z1Z1
Value: 10000
Format no.: 4

Mnemonic: Z0ZMS
Value: 01000
Format no.: 4

Mnemonic: ZMcZS
Value: 00100
Format no.: 4

Mnemonic: ZMnZS
Value: 10100
Format no.: 4

Mnemonic: Z0ZS
Value: 01100
Format no.: 4

Mnemonic: Z0ZSQ
Value: 11100
Format no.: 4

Mnemonic: ZSZQS
Value: 00010
Format no.: 4

Mnemonic: ZMZQS
Value: 10010
Format no.: 4

Mnemonic: ZSZQ
Value: 01010
Format no.: 4

Mnemonic: Z1ZS
Value: 11010
Format no.: 4

Mnemonic: ZMZSQ
Value: 00110
Format no.: 4

Mnemonic: ZQZSQ
Value: 10110
Format no.: 4

Mnemonic: ZI+IZS
Value: 01110
Format no.: 4

Mnemonic: ZQZS
Value: 11110
Format no.: 4

Mnemonic: 0Z0ZS
Value: 00001
Format no.: 4

Mnemonic: 1Z1ZS
Value: 10001
Format no.: 4

Mnemonic: 0Z0Z
Value: 01001
Format no.: 4

Mnemonic: 1Z1Z
Value: 11001
Format no.: 4

Mnemonic: QZ0ZS
Value: 00101
Format no.: 4

Mnemonic: QZ1ZS
Value: 10101
Format no.: 4

Mnemonic: QZØZ
Value: Ø1101
Format no.: 4

Mnemonic: QZ1Z
Value: 11101
Format no.: 4

Mnemonic: SZQZS
Value: ØØØ11
Format no.: 4

Mnemonic: MZQZS
Value: 1ØØ11
Format no.: 4

Mnemonic: SZQZ
Value: Ø1Ø11
Format no.: 4

Mnemonic: MZØZ
Value: 11Ø11
Format no.: 4

Mnemonic: QZMZS
Value: ØØ111
Format no.: 4

Mnemonic: QZSZS
Value: 1Ø111
Format no.: 4

Mnemonic: QZMZ
Value: Ø1111
Format no.: 4

Mnemonic: QZSZ
Value: 11111
Format no.: 4

```
Mnemonic:   Z1ZS1
Value:      11Ø01
Format no.: 4

Mnemonic:   JUMPADDR
Value:      N
Format no.: 12

Mnemonic:   INCR
Value:      Ø
Format no.: 13

Mnemonic:   NOINCR
Value:      1
Format no.: 13

Mnemonic:   NOBRN
Value:      ØØ
Format no.: 14

Mnemonic:   COND
Value:      1Ø
Format no.: 14

Mnemonic:   UNCOND
Value:      Ø1
Format no.: 14

Mnemonic:   CONST
Value:      11
Format no.: 14

Mnemonic:   NOACES
Value:      ØØØ
Format no.: 15

Mnemonic:   DR-DB
Value:      ØØ1
Format no.: 15

Mnemonic:   YB-AR
Value:      Ø1Ø
Format no.: 15
```

```
Mnemonic:   YB-DR
Value:      011
Format no.: 15

Mnemonic:   DR-UIR
Value:      100
Format no.: 15

Mnemonic:   MPC-AR
Value:      101
Format no.: 15

Mnemonic:   DR-MIR
Value:      110
Format no.: 15

Mnemonic:   DR-AR
Value:      111
Format no.: 15

Mnemonic:   CN
Value:      0
Format no.: 16

Mnemonic:   CY
Value:      1
Format no.: 16

Mnemonic:   u0
Value:      000010
Format no.: 10

Mnemonic:   u1
Value:      100010
Format no.: 10

Mnemonic:   u2
Value:      010010
Format no.: 10

Mnemonic:   u3
Value:      110010
Format no.: 10
```

```
Mnemonic:   u4
Value:      001010
Format no.: 10

Mnemonic:   u5
Value:      101010
Format no.: 10

Mnemonic:   u6
Value:      011010
Format no.: 10

Mnemonic:   u7
Value:      111010
Format no.: 10

Mnemonic:   u8
Value:      000110
Format no.: 10

Mnemonic:   u9
Value:      100110
Format no.: 10

Mnemonic:   uA
Value:      010110
Format no.: 10

Mnemonic:   uB
Value:      110110
Format no.: 10

Mnemonic:   uC
Value:      001110
Format no.: 10

Mnemonic:   uD
Value:      101110
Format no.: 10

Mnemonic:   uE
Value:      011110
Format no.: 10
```

```
Mnemonic:   uF
Value:      111110
Format no.: 10

Mnemonic:   M0
Value:      000001
Format no.: 10

Mnemonic:   M1
Value:      100001
Format no.: 10

Mnemonic:   M2
Value:      010001
Format no.: 10

Mnemonic:   M3
Value:      110001
Format no.: 10

Mnemonic:   M4
Value:      001001
Format no.: 10

Mnemonic:   M5
Value:      101001
Format no.: 10

Mnemonic:   M6
Value:      011001
Format no.: 10

Mnemonic:   M7
Value:      111001
Format no.: 10

Mnemonic:   M8
Value:      000101
Format no.: 10

Mnemonic:   M9
Value:      100101
Format no.: 10
```

Mnemonic: MA
Value: 010101
Format no.: 10

Mnemonic: MB
Value: 110101
Format no.: 10

Mnemonic: MC
Value: 001101
Format no.: 10

Mnemonic: MD
Value: 101101
Format no.: 10

Mnemonic: ME
Value: 011101
Format no.: 10

Mnemonic: MF
Value: 111101
Format no.: 10

Mnemonic: KF
Value: 11
Format no.: 17

File : proto1

LISTING OF THE TABLE FILE
==========

Mnemonic: instrlengt
Value: 56
Format: 0

Mnemonic: AQ
Value: 000
Format: 39x 3a 14x

Mnemonic: AB
Value: 100
Format: 39x 3a 14x

Mnemonic: ZQ
Value: 010
Format: 39x 3a 14x

Mnemonic: ZB
Value: 110
Format: 39x 3a 14x

Mnemonic: ZA
Value: 001
Format: 39x 3a 14x

Mnemonic: DA
Value: 101
Format: 39x 3a 14x

Mnemonic: DQ
Value: 011
Format: 39x 3a 14x

Mnemonic: DZ
Value: 111
Format: 39x 3a 14x

Mnemonic: ADD
Value: 000
Format: 42x 3a 11x

```
Mnemonic:   SUBR
Value:      100
Format:     42x 3a 11x

Mnemonic:   SUBS
Value:      010
Format:     42x 3a 11x

Mnemonic:   OR
Value:      110
Format:     42x 3a 11x

Mnemonic:   AND
Value:      001
Format:     42x 3a 11x

Mnemonic:   NOTRS
Value:      101
Format:     42x 3a 11x

Mnemonic:   EXOR
Value:      011
Format:     42x 3a 11x

Mnemonic:   EXNOR
Value:      111
Format:     42x 3a 11x

Mnemonic:   QREG
Value:      000
Format:     45x 3a 8x

Mnemonic:   NOP
Value:      100
Format:     45x 3a 8x

Mnemonic:   RAMA
Value:      010
Format:     45x 3a 8x

Mnemonic:   RAMF
Value:      110
Format:     45x 3a 8x
```

```
Mnemonic:   RAMQD
Value:      ØØ1
Format:     45x 3a 8x

Mnemonic:   RAMD
Value:      1Ø1
Format:     45x 3a 8x

Mnemonic:   RAMQU
Value:      Ø11
Format:     45x 3a 8x

Mnemonic:   RAMU
Value:      111
Format:     45x 3a 8x

Mnemonic:   Areg
Value:      n
Format:     48x 4a 4x

Mnemonic:   Breg
Value:      n
Format:     52x 4a

Mnemonic:   Ø-UZ
Value:      ØØØ1ØØØ1
Format:     24x 6a 7x 2a 17x

Mnemonic:   1-UZ
Value:      1ØØ1ØØØ1
Format:     24x 6a 7x 2a 17x

Mnemonic:   Ø-UC
Value:      Ø1Ø1ØØØ1
Format:     24x 6a 7x 2a 17x

Mnemonic:   1-UC
Value:      11Ø1ØØØ1
Format:     24x 6a 7x 2a 17x

Mnemonic:   Ø-UN
Value:      ØØ11ØØØ1
Format:     24x 6a 7x 2a 17x
```

```
Mnemonic:   1-UN
Value:      10110001
Format:     24x 6a 7x 2a 17x

Mnemonic:   0-UOVR
Value:      01110001
Format:     24x 6a 7x 2a 17x

Mnemonic:   1-UOVR
Value:      11110001
Format:     24x 6a 7x 2a 17x

Mnemonic:   MX-UX
Value:      00000001
Format:     24x 6a 7x 2a 17x

Mnemonic:   1-UX
Value:      10000001
Format:     24x 6a 7x 2a 17x

Mnemonic:   MX+UX
Value:      01000001
Format:     24x 6a 7x 2a 17x

Mnemonic:   0-UX
Value:      11000001
Format:     24x 6a 7x 2a 17x

Mnemonic:   I-UO
Value:      11100001
Format:     24x 6a 7x 2a 17x

Mnemonic:   I-UC
Value:      00011001
Format:     24x 6a 7x 2a 17x

Mnemonic:   I-U
Value:      00100001
Format:     24x 6a 7x 2a 17x

Mnemonic:   YX-MX
Value:      00000010
Format:     24x 6a 7x 2a 17x
```

```
Mnemonic:   1-MX
Value:      10000010
Format:     24x 6a 7x 2a 17x

Mnemonic:   UX-MX
Value:      01000010
Format:     24x 6a 7x 2a 17x

Mnemonic:   0-MX
Value:      11000010
Format:     24x 6a 7x 2a 17x

Mnemonic:   MX-MX
Value:      10100010
Format:     24x 6a 7x 2a 17x

Mnemonic:   I-MO
Value:      00100010
Format:     24x 6a 7x 2a 17x

Mnemonic:   I-MC
Value:      00010010
Format:     24x 6a 7x 2a 17x

Mnemonic:   I-M
Value:      11110010
Format:     24x 6a 7x 2a 17x

Mnemonic:   CO=0
Value:      00
Format:     35x 2a 19x

Mnemonic:   CO=1
Value:      10
Format:     35x 2a 19x

Mnemonic:   CO=CX
Value:      01
Format:     35x 2a 19x

Mnemonic:   CO=UC
Value:      0000011
Format:     25x 5a 5x 2a 19x
```

```
Mnemonic:   CO=NUC
Value:      0010011
Format:     25x 5a 5x 2a 19x

Mnemonic:   CO=MC
Value:      0000111
Format:     25x 5a 5x 2a 19x

Mnemonic:   CO=NMC
Value:      0010111
Format:     25x 5a 5x 2a 19x

Mnemonic:   Z0Z0
Value:      00000
Format:     30x 5a 21x

Mnemonic:   Z1Z1
Value:      10000
Format:     30x 5a 21x

Mnemonic:   Z0ZMS
Value:      01000
Format:     30x 5a 21x

Mnemonic:   ZMcZS
Value:      00100
Format:     30x 5a 21x

Mnemonic:   ZMnZS
Value:      10100
Format:     30x 5a 21x

Mnemonic:   Z0ZS
Value:      01100
Format:     30x 5a 21x

Mnemonic:   Z0ZSQ
Value:      11100
Format:     30x 5a 21x

Mnemonic:   ZSZQS
Value:      00010
Format:     30x 5a 21x
```

```
Mnemonic:   ZMZQS
Value:      10010
Format:     30x 5a 21x

Mnemonic:   ZSZQ
Value:      01010
Format:     30x 5a 21x

Mnemonic:   Z1ZS
Value:      11010
Format:     30x 5a 21x

Mnemonic:   ZMZSQ
Value:      00110
Format:     30x 5a 21x

Mnemonic:   ZQZSQ
Value:      10110
Format:     30x 5a 21x

Mnemonic:   ZI+IZS
Value:      01110
Format:     30x 5a 21x

Mnemonic:   ZQZS
Value:      11110
Format:     30x 5a 21x

Mnemonic:   0Z0ZS
Value:      00001
Format:     30x 5a 21x

Mnemonic:   1Z1ZS
Value:      10001
Format:     30x 5a 21x

Mnemonic:   0Z0Z
Value:      01001
Format:     30x 5a 21x

Mnemonic:   1Z1Z
Value:      11001
Format:     30x 5a 21x
```

Mnemonic: QZ0ZS
Value: 00101
Format: 30x 5a 21x

Mnemonic: QZ1ZS
Value: 10101
Format: 30x 5a 21x

Mnemonic: QZ0Z
Value: 01101
Format: 30x 5a 21x

Mnemonic: QZ1Z
Value: 11101
Format: 30x 5a 21x

Mnemonic: SZQZS
Value: 00011
Format: 30x 5a 21x

Mnemonic: MZQZS
Value: 10011
Format: 30x 5a 21x

Mnemonic: SZQZ
Value: 01011
Format: 30x 5a 21x

Mnemonic: MZ0Z
Value: 11011
Format: 30x 5a 21x

Mnemonic: QZMZS
Value: 00111
Format: 30x 5a 21x

Mnemonic: QZSZS
Value: 10111
Format: 30x 5a 21x

Mnemonic: QZMZ
Value: 01111
Format: 30x 5a 21x

```
Mnemonic:   QZSZ
Value:      11111
Format:     30x 5a 21x

Mnemonic:   Z1ZS1
Value:      11001
Format:     30x 5a 21x

Mnemonic:   JUMPADDR
Value:      N
Format:     12a 44x

Mnemonic:   INCR
Value:      0
Format:     12x 1a 43x

Mnemonic:   NOINCR
Value:      1
Format:     12x 1a 43x

Mnemonic:   NOBRN
Value:      00
Format:     13x 2a 41x

Mnemonic:   COND
Value:      10
Format:     13x 2a 41x

Mnemonic:   UNCOND
Value:      01
Format:     13x 2a 41x

Mnemonic:   CONST
Value:      11
Format:     13x 2a 41x

Mnemonic:   NOACES
Value:      000
Format:     15x 3a 38x

Mnemonic:   DR-DB
Value:      001
Format:     15x 3a 38x
```

```
Mnemonic:   YB-AR
Value:      010
Format:     15x 3a 38x

Mnemonic:   YB-DR
Value:      011
Format:     15x 3a 38x

Mnemonic:   DR-UIR
Value:      100
Format:     15x 3a 38x

Mnemonic:   MPC-AR
Value:      101
Format:     15x 3a 38x

Mnemonic:   DR-MIR
Value:      110
Format:     15x 3a 38x

Mnemonic:   DR-AR
Value:      111
Format:     15x 3a 38x

Mnemonic:   CN
Value:      0
Format:     18x 1a 37x

Mnemonic:   CY
Value:      1
Format:     18x 1a 37x

Mnemonic:   u0
Value:      000010
Format:     24x 6a 26x

Mnemonic:   u1
Value:      100010
Format:     24x 6a 26x

Mnemonic:   u2
Value:      010010
Format:     24x 6a 26x
```

```
Mnemonic:   u3
Value:      110010
Format:     24x 6a 26x

Mnemonic:   u4
Value:      001010
Format:     24x 6a 26x

Mnemonic:   u5
Value:      101010
Format:     24x 6a 26x

Mnemonic:   u6
Value:      011010
Format:     24x 6a 26x

Mnemonic:   u7
Value:      111010
Format:     24x 6a 26x

Mnemonic:   u8
Value:      000110
Format:     24x 6a 26x

Mnemonic:   u9
Value:      100110
Format:     24x 6a 26x

Mnemonic:   uA
Value:      010110
Format:     24x 6a 26x

Mnemonic:   uB
Value:      110110
Format:     24x 6a 26x

Mnemonic:   uC
Value:      001110
Format:     24x 6a 26x

Mnemonic:   uD
Value:      101110
Format:     24x 6a 26x
```

```
Mnemonic:   uE
Value:      011110
Format:     24x 6a 26x

Mnemonic:   uF
Value:      111110
Format:     24x 6a 26x

Mnemonic:   M0
Value:      000001
Format:     24x 6a 26x

Mnemonic:   M1
Value:      100001
Format:     24x 6a 26x

Mnemonic:   M2
Value:      010001
Format:     24x 6a 26x

Mnemonic:   M3
Value:      110001
Format:     24x 6a 26x

Mnemonic:   M4
Value:      001001
Format:     24x 6a 26x

Mnemonic:   M5
Value:      101001
Format:     24x 6a 26x

Mnemonic:   M6
Value:      011001
Format:     24x 6a 26x

Mnemonic:   M7
Value:      111001
Format:     24x 6a 26x

Mnemonic:   M8
Value:      000101
Format:     24x 6a 26x
```

```
Mnemonic:   M9
Value:      100101
Format:     24x 6a 26x

Mnemonic:   MA
Value:      010101
Format:     24x 6a 26x

Mnemonic:   MB
Value:      110101
Format:     24x 6a 26x

Mnemonic:   MC
Value:      001101
Format:     24x 6a 26x

Mnemonic:   MD
Value:      101101
Format:     24x 6a 26x

Mnemonic:   ME
Value:      011101
Format:     24x 6a 26x

Mnemonic:   MF
Value:      111101
Format:     24x 6a 26x

Mnemonic:   KF
Value:      11
Format:     37x 2a 17x
```

File : proto1

LISTING OF THE MACRO-INSTRUCTION FILE
==========

```
Macro-instruction:   fetch
Hexadecimal opcode:
Micro-codes:
Areg #0000 & Breg #0000 & RAMA & ZB & ADD & CO=1 & I-U & YB-AR & CY
NOP & DR-MIR & NOINCR & KF


Macro-instruction:   LDA
Hexadecimal opcode:  11
Micro-codes:
Areg #0000 & Breg #0000 & RAMA & ZB & ADD & CO=1 & I-U & YB-AR & CY
NOP & DR-AR & KF
DR-DB & DZ & OR & Breg #0001 & RAMF & I-M & CY
NOP & JUMPADDR # fetch & NOINCR & UNCOND & KF


Macro-instruction:   LDA#
Hexadecimal opcode:  10
Micro-codes:
Areg #0000 & Breg #0000 & RAMA & ZB & ADD & CO=1 & I-U & YB-AR & CY
DR-DB & DZ & OR & Breg #0001 & RAMF & I-M & CY
NOP & JUMPADDR # fetch & NOINCR & UNCOND & KF


Macro-instruction:   STA
Hexadecimal opcode:  20
Micro-codes:
Areg #0000 & Breg #0000 & RAMA & ZB & ADD & CO=1 & I-U & YB-AR & CY
NOP & DR-AR & KF
Breg #0001 & ZB & OR & NOP & I-M & YB-DR & CY
NOP & JUMPADDR # fetch & NOINCR & UNCOND & KF


Macro-instruction:   ADD#
Hexadecimal opcode:  30
Micro-codes:
Areg #0000 & Breg #0000 & RAMA & ZB & ADD & CO=1 & I-U & YB-AR & CY
DR-DB & Breg #0001 & Areg #0001 & DA & ADD & RAMF & I-M & CY
NOP & JUMPADDR # fetch & NOINCR & UNCOND & KF


Macro-instruction:   SUB#
Hexadecimal opcode:  40
Micro-codes:
```

```
Areg #0000 & Breg #0000 & RAMA & ZB & ADD & CO=1 & I-U & YB-AR & CY
DR-DB & Breg #0001 & Areg #0001 & DA & SUBR & RAMF & I-M & CO=1 & CY
NOP & JUMPADDR #fetch & NOINCR & UNCOND & KF
```

```
Macro-instruction:   JMP
Hexadecimal opcode:  60
Micro-codes:
Areg #0000 & Breg #0000 & RAMA & ZB & ADD & CO=1 & I-U & YB-AR & CY
DR-DB & Breg #0000 & DZ & ADD & RAMF & I-U & CY
NOP & JUMPADDR #fetch & NOINCR & UNCOND & KF
```

```
Macro-instruction:   JMZ
Hexadecimal opcode:  61
Micro-codes:
Areg #0000 & Breg #0000 & RAMA & ZB & ADD & CO=1 & I-U & YB-AR & CY
Areg #0000 & Breg #1111 & ZA & OR & RAMF & I-U & CY
DR-DB & Areg #0000 & Breg #0000 & DZ & ADD & RAMF & I-U & CY
NOP & M4 & CY & KF
NOP & JUMPADDR #fetch & NOINCR & COND & KF
Areg #1111 & Breg #0000 & ZA & OR & RAMF & I-U & CY
NOP & JUMPADDR #fetch & NOINCR & UNCOND & KF
```

```
Macro-instruction:   startup
Hexadecimal opcode:
Micro-codes:
;this is the general startup routine which should be executed first
Breg #0000 & DZ & OR & RAMF & I-U & CY ;assume the PC is on the D bus
NOP & JUMPADDR # fetch & NOINCR & UNCOND & KF
```

```
Macro-instruction:   AND#
Hexadecimal opcode:  51
Micro-codes:
Areg #0000 & Breg #0000 & RAMA & ZB & ADD & CO=1 & I-U & YB-AR & CY
DR-DB & Breg #0001 & Areg #0001 & DA & AND & RAMF & I-M & CY
NOP & JUMPADDR # fetch & NOINCR & UNCOND & KF
```

```
Macro-instruction:   OR#
Hexadecimal opcode:  52
Micro-codes:
Areg #0000 & Breg #0000 & RAMA & ZB & ADD & CO=1 & I-U & YB-AR & CY
DR-DB & Breg #0001 & Areg #0001 & DA & OR & RAMF & I-M & CY
NOP & JUMPADDR # fetch & NOINCR & UNCOND & KF
```

```
Macro-instruction:   INL
Hexadecimal opcode:  01
Micro-codes:
Areg #0000 & Breg #0000 & RAMA & ZB & ADD & CO=1 & I-U & YB-AR & CY
DR-UIR & JUMPADDR #000000000000 & KF
Areg #0000 & Breg #0000 & RAMA & ZB & ADD & CO=1 & I-U & YB-AR & CY
DR-UIR & JUMPADDR #000000000001 & KF  ; DR-LAT
Areg #0000 & Breg #0000 & RAMA & ZB & ADD & CO=1 & I-U & YB-AR & CY
DR-UIR & JUMPADDR #000000000010 & KF  ; DR-LAT
Areg #0000 & Breg #0000 & RAMA & ZB & ADD & CO=1 & I-U & YB-AR & CY
DR-UIR & JUMPADDR #000000000011 & KF  ; DR-LAT
MPC-AR & KF & NOINCR  ; LAT-UIR
NOP & JUMPADDR # fetch & NOINCR & UNCOND & KF


Macro-instruction:   NOT
Hexadecimal opcode:  50
Micro-codes:
Breg #0001 & ZB & EXNOR & RAMF & I-M & CY
NOP & JUMPADDR #fetch & NOINCR & UNCOND & KF


Macro-instruction:   JMNZ
Hexadecimal opcode:  62
Micro-codes:
Areg #0000 & Breg #0000 & RAMA & ZB & ADD & CO=1 & I-U & YB-AR & CY
Areg #0000 & Breg #1111 & ZA & OR & RAMF & I-U & CY
DR-DB & Areg #0000 & Breg #0000 & DZ & ADD & RAMF & I-U & CY
NOP & M5 & CY & KF
NOP & JUMPADDR #fetch & NOINCR & COND & KF
Areg #1111 & Breg #0000 & ZA & OR & RAMF & I-U & CY
NOP & JUMPADDR #fetch & NOINCR & UNCOND & KF


Macro-instruction:   JMNEG
Hexadecimal opcode:  63
Micro-codes:
Areg #0000 & Breg #0000 & RAMA & ZB & ADD & CO=1 & I-U & YB-AR & CY
Areg #0000 & Breg #1111 & ZA & OR & RAMF & I-U & CY
DR-DB & Areg #0000 & Breg #0000 & DZ & ADD & RAMF & I-U & CY
NOP & ME & CY & KF
NOP & JUMPADDR #fetch & NOINCR & COND & KF
Areg #1111 & Breg #0000 & ZA & OR & RAMF & I-U & CY
NOP & JUMPADDR #fetch & NOINCR & UNCOND & KF


Macro-instruction:   ADD
Hexadecimal opcode:  31
Micro-codes:
Areg #0000 & Breg #0000 & RAMA & ZB & ADD & CO=1 & I-U & YB-AR & CY
NOP & DR-AR & KF
DR-DB & Breg #0001 & Areg #0001 & DA & ADD & RAMF & I-M & CY
NOP & JUMPADDR #fetch & NOINCR & UNCOND & KF
```

```
Macro-instruction:   SUB
Hexadecimal opcode:  41
Micro-codes:
Areg #0000 & Breg #0000 & RAMA & ZB & ADD & CO=1 & I-U & YB-AR & CY
NOP & DR-AR & KF
DR-DB & Breg #0001 & Areg #0001 & DA & SUBR & RAMF & I-M & CO=1 & CY
NOP & JUMPADDR #fetch & NOINCR & UNCOND & KF


Macro-instruction:   JMOVR
Hexadecimal opcode:  64
Micro-codes:
Areg #0000 & Breg #0000 & RAMA & ZB & ADD & CO=1 & I-U & YB-AR & CY
Areg #0000 & Breg #1111 & ZA & OR & RAMF & I-U & CY
DR-DB & Areg #0000 & Breg #0000 & DZ & ADD & RAMF & I-U & CY
NOP & M6 & CY & KF
NOP & JUMPADDR #fetch & NOINCR & COND & KF
Areg #1111 & Breg #0000 & ZA & OR & RAMF & I-U & CY
NOP & JUMPADDR #fetch & NOINCR & UNCOND & KF


Macro-instruction:   AND
Hexadecimal opcode:  53
Micro-codes:
Areg #0000 & Breg #0000 & RAMA & ZB & ADD & CO=1 & I-U & YB-AR & CY
NOP & DR-AR & KF
DR-DB & Breg #0001 & Areg #0001 & DA & AND & RAMF & I-M & CY
NOP & JUMPADDR #fetch & NOINCR & UNCOND & KF


Macro-instruction:   SHL
Hexadecimal opcode:  70
Micro-codes:
Breg #0001 & ZB & OR & RAMU & 0Z0Z & I-M & CY
NOP & JUMPADDR #fetch & NOINCR & UNCOND & KF


Macro-instruction:   SHR
Hexadecimal opcode:  71
Micro-codes:
Breg #0001 & ZB & OR & RAMD & Z0Z0 & I-M & CY
NOP & JUMPADDR #fetch & NOINCR & UNCOND & KF


Macro-instruction:   CMP#
Hexadecimal opcode:  80
Micro-codes:
Areg #0000 & Breg #0000 & RAMA & ZB & ADD & CO=1 & I-U & YB-AR & CY
DR-DB & Areg #0001 & DA & SUBR & NOP & I-M & CO=1 & CY
NOP & JUMPADDR #fetch & NOINCR & UNCOND & KF
```

```
Macro-instruction:   CMP
Hexadecimal opcode:  81
Micro-codes:
Areg #0000 & Breg #0000 & RAMA & ZB & ADD & CO=1 & I-U & YB-AR & CY
NOP & DR-AR & KF
DR-DB & Areg #0001 & DA & SUBR & NOP & I-M & CO=1 & CY
NOP & JUMPADDR #fetch & NOINCR & UNCOND & KF
```

```
Macro-instruction:   OR
Hexadecimal opcode:  54
Micro-codes:
Areg #0000 & Breg #0000 & RAMA & ZB & ADD & CO=1 & I-U & YB-AR & CY
NOP & DR-AR & KF
DR-DB & Breg #0001 & Areg #0001 & DA & OR & RAMF & I-M & CY
NOP & JUMPADDR #fetch & NOINCR & UNCOND & KF
```

File : proto1

LISTING OF THE DECODE FILE
==========

```
Mnemonic : fetch    Opcode :     Address : 0
Mnemonic : LDA    Opcode : 11    Address : 2
Mnemonic : LDA#    Opcode : 10    Address : 6
Mnemonic : STA    Opcode : 20    Address : 9
Mnemonic : ADD#    Opcode : 30    Address : 13
Mnemonic : SUB#    Opcode : 40    Address : 16
Mnemonic : JMP    Opcode : 60    Address : 19
Mnemonic : JMZ    Opcode : 61    Address : 22
Mnemonic : startup    Opcode :     Address : 29
Mnemonic : AND#    Opcode : 51    Address : 31
Mnemonic : OR#    Opcode : 52    Address : 34
Mnemonic : INL    Opcode : 01    Address : 37
Mnemonic : NOT    Opcode : 50    Address : 47
Mnemonic : JMNZ    Opcode : 62    Address : 49
Mnemonic : JMNEG    Opcode : 63    Address : 56
Mnemonic : ADD    Opcode : 31    Address : 63
Mnemonic : SUB    Opcode : 41    Address : 67
Mnemonic : JMOVR    Opcode : 64    Address : 71
Mnemonic : AND    Opcode : 53    Address : 78
Mnemonic : SHL    Opcode : 70    Address : 82
Mnemonic : SHR    Opcode : 71    Address : 84
Mnemonic : CMP#    Opcode : 80    Address : 86
Mnemonic : CMP    Opcode : 81    Address : 89
Mnemonic : OR    Opcode : 54    Address : 93
```

```
File : proto1

LISTING OF THE MICRO-CODE FILE
==============================

fetch
0000A020138200
0009800006040Ø

LDA   11
0000A020138200
0001C000060400
000060F005F601
000A0000060400

LDA#   10
0000A020138200
000060F005F601
000A0000060400

STA   20
0000A020138200
0001C000060400
0000E0F005B401
000A0000060400

ADD#   30
0000A020138200
000060F0054611
000A0000060400

SUB#   40
0000A020138200
000060F0156611
000A0000060400

JMP   60
0000A020138200
0000602003C600
000A0000060400

JMZ   61
0000A020138200
0000202002760F
0000602003C600
0000202406040Ø
000C0000060400
0000202002760Ø
000A0000060400

startup
0000202003F600
000A0000060400

AND#   51
```

```
0000A020138200
000060F0054E11
000A0000060400

OR#   52
0000A020138200
000060F0057611
000A0000060400

INL   01
0000A020138200
0001000006000Ø
0000A020138200
0011000006000Ø
0000A020138200
0021000006000Ø
0000A020138200
0031000006000Ø
0009400006000Ø
000A0000060400

NOT   50
000020F005BE01
000A0000060400

JMNZ   62
0000A020138200
0000202002760F
0000602003C600
000020A4060400
000C0000060400
0000202002760F0
000A0000060400

JMNEG   63
0000A020138200
0000202002760F
0000602003C600
0000207406Ø400
000C0000060400
0000202002760F0
000A0000060400

ADD   31
0000A020138200
0001C000060400
000060F0054611
000A0000060400

SUB   41
0000A020138200
0001C000060400
000060F0156611
000A0000060400
```

```
JMOVR   64
0000A020138200
0000202002760F
0000602003C600
0000206406040Ø
000C0000060400
000020200276FØ
000A0000060400

AND   53
0000A020138200
0001C000060400
000060F0054E11
000A0000060400

SHL   7Ø
000020F125B7Ø1
000A0000060400

SHR   71
000020F005B501
000A0000060400

CMP#   8Ø
0000A020138200
000060F0156410
000A0000060400

CMP   81
0000A020138200
0001C000060400
000060F0156410
000A0000060400

OR   54
0000A020138200
0001C000060400
000060F0057611
000A0000060400
```

File : proto1

LISTING OF THE MICRO-CODE BINARY FILE
==========

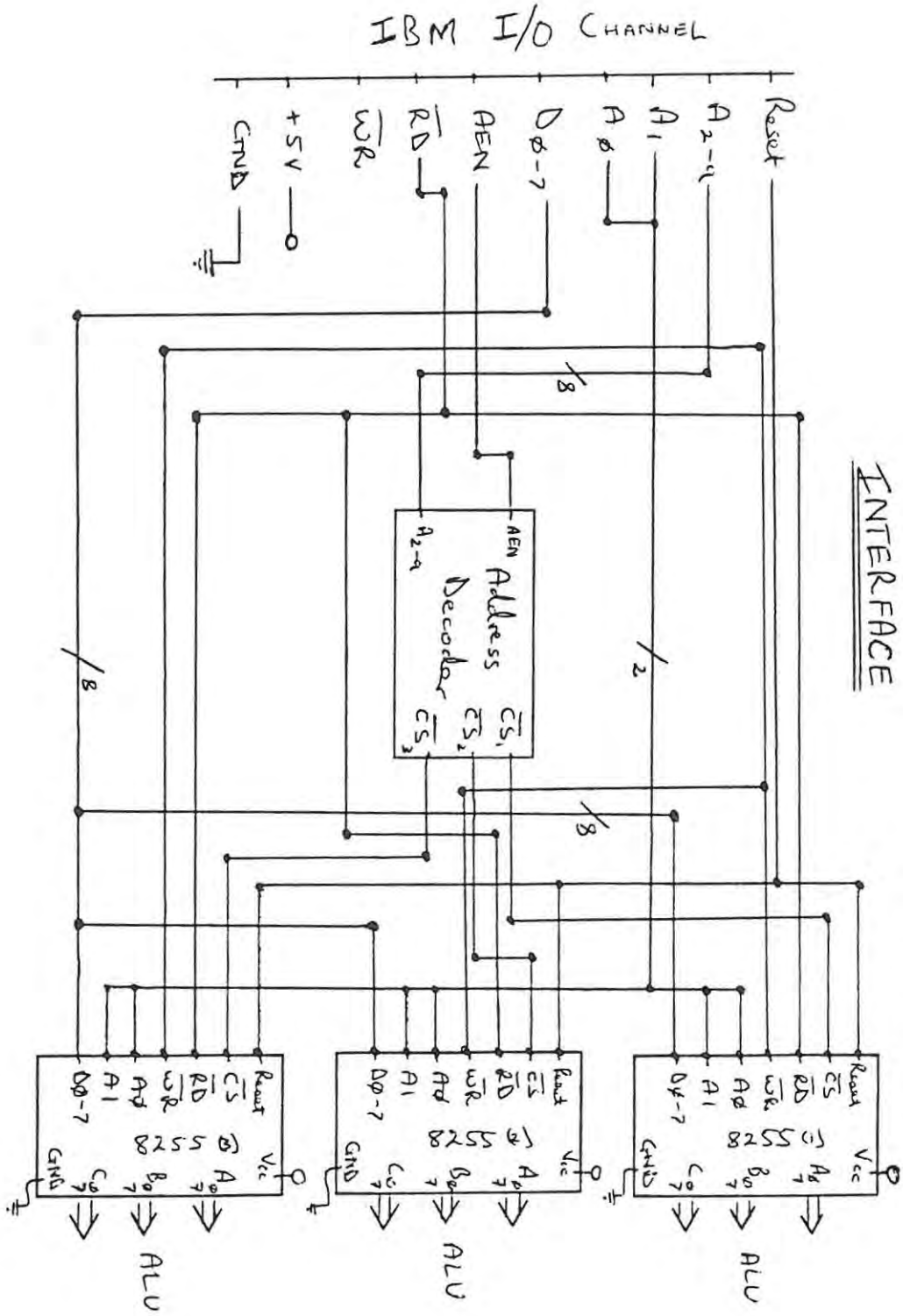
```
0  00000000000000001010000000100000000100111000001000000000
1  00000000000010011000000000000000000001100000010000000000
2  00000000000000001010000000100000000100111000001000000000
3  00000000000000011100000000000000000001100000010000000000
4  00000000000000001100000111100000000010111110110000000001
5  00000000000010100000000000000000000001100000010000000000
6  00000000000000001010000000100000000100111000001000000000
7  00000000000000001100000111100000000010111110110000000001
8  00000000000010100000000000000000000001100000010000000000
9  00000000000000001010000000100000000100111000001000000000
10  00000000000000011100000000000000000001100000010000000000
11  00000000000000001110000111100000000010110110100000000001
12  00000000000010100000000000000000000001100000010000000000
13  00000000000000001010000000100000000100111000001000000000
14  00000000000000001100000111100000000010101000110000100001
15  00000000000010100000000000000000000001100000010000000000
16  00000000000000001010000000100000000100111000001000000000
17  00000000000000001100000111100000000010101011001100001000 1
18  00000000000010100000000000000000000001100000010000000000
19  00000000000000001010000000100000000100111000001000000000
20  00000000000000001100000001000000000001111000110000000000
21  00000000000010100000000000000000000001100000010000000000
22  00000000000000001010000000100000000100111000001000000000
23  00000000000000000010000000100000000001001110110000011111
24  00000000000000001100000001000000000001111000110000000000
25  00000000000000000010000000100100000001100000010000000000
26  00000000000011000000000000000000000001100000010000000000
27  00000000000000000010000000100000000001001110110111100000
28  00000000000010100000000000000000000001100000010000000000
29  00000000000000000010000000100000000001111110110000000000
30  00000000000010100000000000000000000001100000010000000000
31  00000000000000001010000000100000000100111000001000000000
32  00000000000000001100000111100000000010101001110000100001
33  00000000000010100000000000000000000001100000010000000000
34  00000000000000001010000000100000000100111000001000000000
35  00000000000000001100000111100000000010101110110000100001
36  00000000000010100000000000000000000001100000010000000000
37  00000000000000001010000000100000000100111000001000000000
38  00000000000000001000000000000000000001100000000000000000
39  00000000000000001010000000100000000100111000001000000000
40  00000000000010001000000000000000000001100000000000000000
41  00000000000000001010000000100000000100111000001000000000
42  00000000000100001000000000000000000001100000000000000000
43  00000000000000001010000000100000000100111000001000000000
44  00000000001100001000000000000000000001100000000000000000
45  00000000000010010100000000000000000001100000000000000000
46  00000000000010100000000000000000000001100000010000000000
47  00000000000000001000001111000000000010110111110000000001
48  00000000000010100000000000000000000001100000010000000000
49  00000000000000001010000000100000000100111000001000000000
```

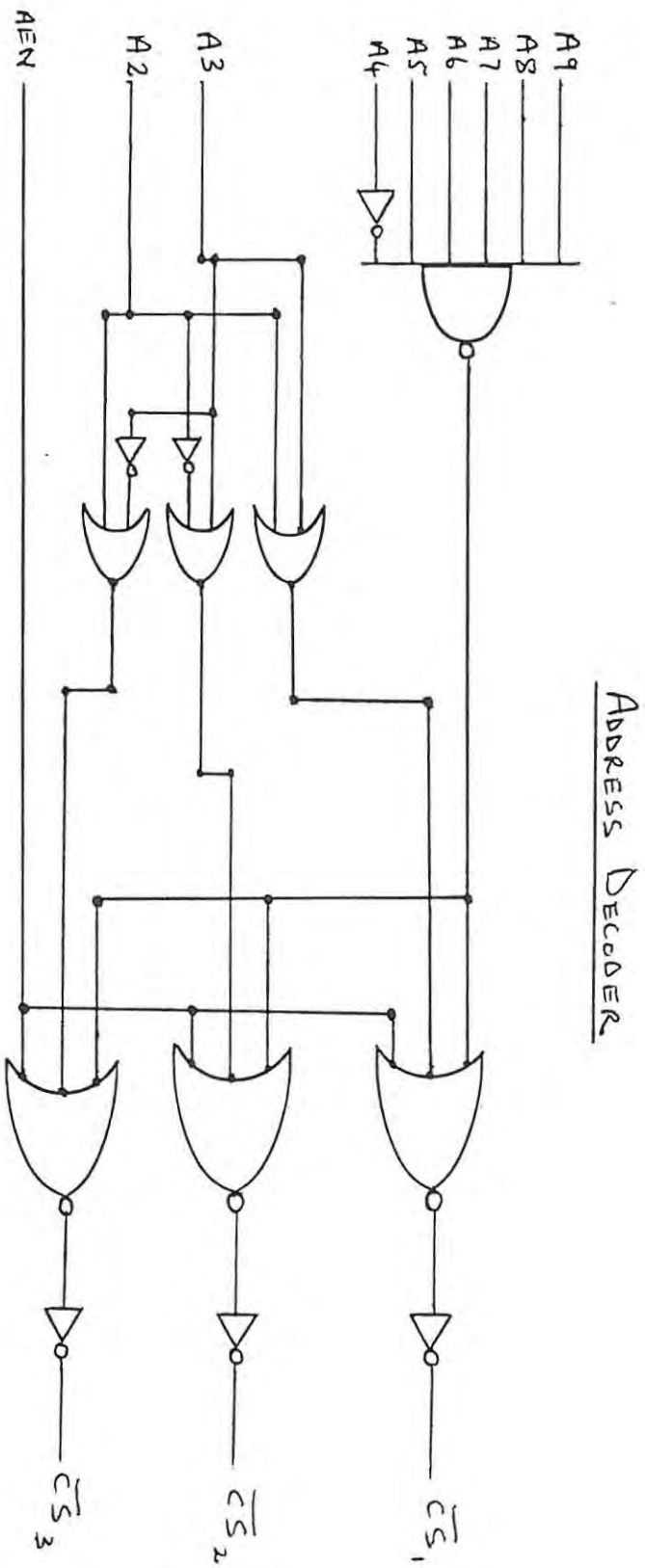
```
50  00000000000000000010000000100000000000100111011000001111
51  00000000000000000011000000100000000000111100011000000000
52  00000000000000000010000010100100000001100000010000000000
53  00000000000011000000000000000000000000110000001000000000
54  00000000000000000010000000100000000000100111011011110000
55  00000000000010100000000000000000000000110000001000000000
56  00000000000000001010000000100000000100111000001000000000
57  00000000000000000010000000100000000000100111011000001111
58  00000000000000001100000001000000000000111100011000000000
59  00000000000000000010000001110100000001100000010000000000
60  00000000000011000000000000000000000000110000001000000000
61  00000000000000000010000000100000000000100111011011110000
62  00000000000010100000000000000000000000110000001000000000
63  00000000000000001010000000100000000100111000001000000000
64  00000000000000001110000000000000000000110000001000000000
65  00000000000000001100000111100000000010101000110000100001
66  00000000000010100000000000000000000000110000001000000000
67  00000000000000001010000000100000000100111000001000000000
68  00000000000000001110000000000000000000110000001000000000
69  00000000000000001100000111100000000010101011001100001000 1
70  00000000000010100000000000000000000000110000001000000000
71  00000000000000001010000000100000000100111000001000000000
72  00000000000000000010000000100000000000100111011000001111
73  00000000000000001100000001000000000000111100011000000000
74  00000000000000000010000001100100000001100000010000000000
75  00000000000011000000000000000000000000110000001000000000
76  00000000000000000010000000100000000000100111011011110000
77  00000000000010100000000000000000000000110000001000000000
78  00000000000000001010000000100000000100111000001000000000
79  00000000000000001110000000000000000000110000001000000000
80  00000000000000001100000111100000000010101001110000100001
81  00000000000010100000000000000000000000110000001000000000
82  00000000000000000010000011110001001001011011011100000001
83  00000000000010100000000000000000000000110000001000000000
84  00000000000000000010000011110000000001011011010100000001
85  00000000000010100000000000000000000000110000001000000000
86  00000000000000001010000000100000000100111000001000000000
87  00000000000000001100000111100000000010101011001000001000 0
88  00000000000010100000000000000000000000110000001000000000
89  00000000000000001010000000100000000100111000001000000000
90  00000000000000001110000000000000000000110000001000000000
91  00000000000000001100000111100000000010101011001000001000 0
92  00000000000010100000000000000000000000110000001000000000
93  00000000000000001010000000100000000100111000001000000000
94  00000000000000001110000000000000000000110000001000000000
95  00000000000000001100000111100000000010101110110000100001
96  00000000000010100000000000000000000000110000001000000000
```

Appendix H

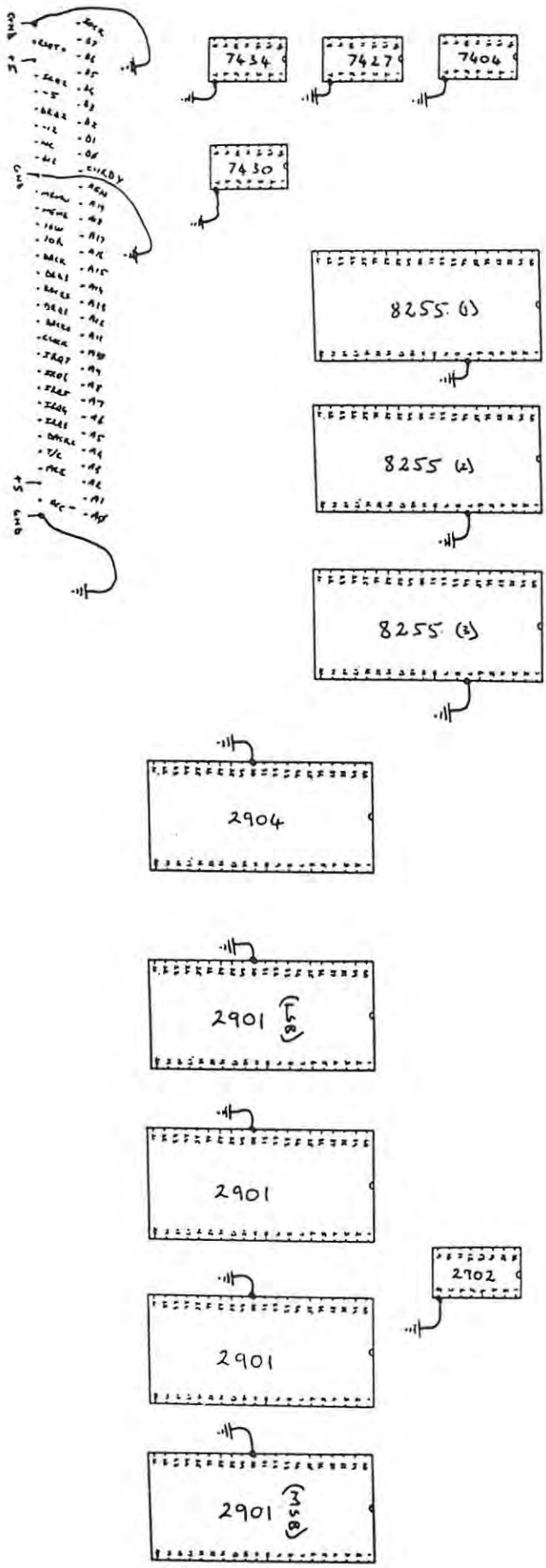
Appendix H.1 contains the schematic circuit diagrams for the arithmetic and logic unit hardware interface. The ALU circuits themselves are connected in the standard way suggested in the Am2900 family data book [ADV].

Appendix H.2 gives a step by step breakdown of the wire wrapped circuit board. These diagrams should enable the user to perform necessary maintenance and should also provide the necessary information for the production of further boards.


IBM I/O CHANNEL
Reset
A2-9
A1
A0
DØ-7
AEN
RD
WR
+5V
GND
8
2
Address Decoder
AEN
A2-9
CS1
CS2
CS3
INTERFACE
8255
ALU



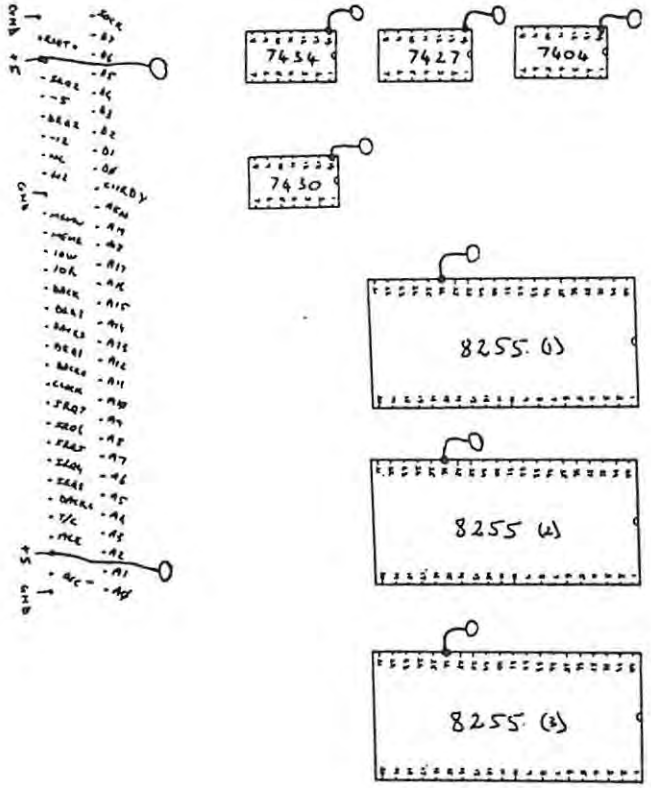
Address Decoder

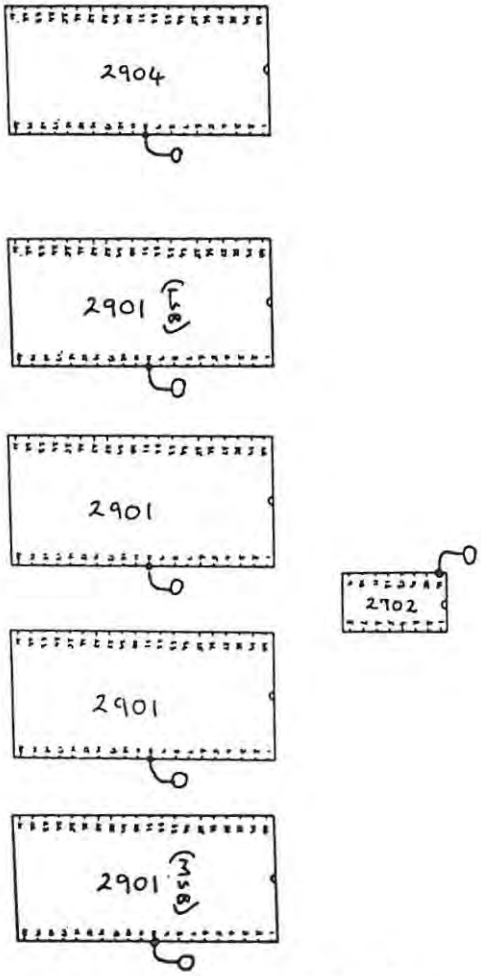


Wire Wrapped Side

Circuit Section: Power Supply (earth)

Wire Colour: Black

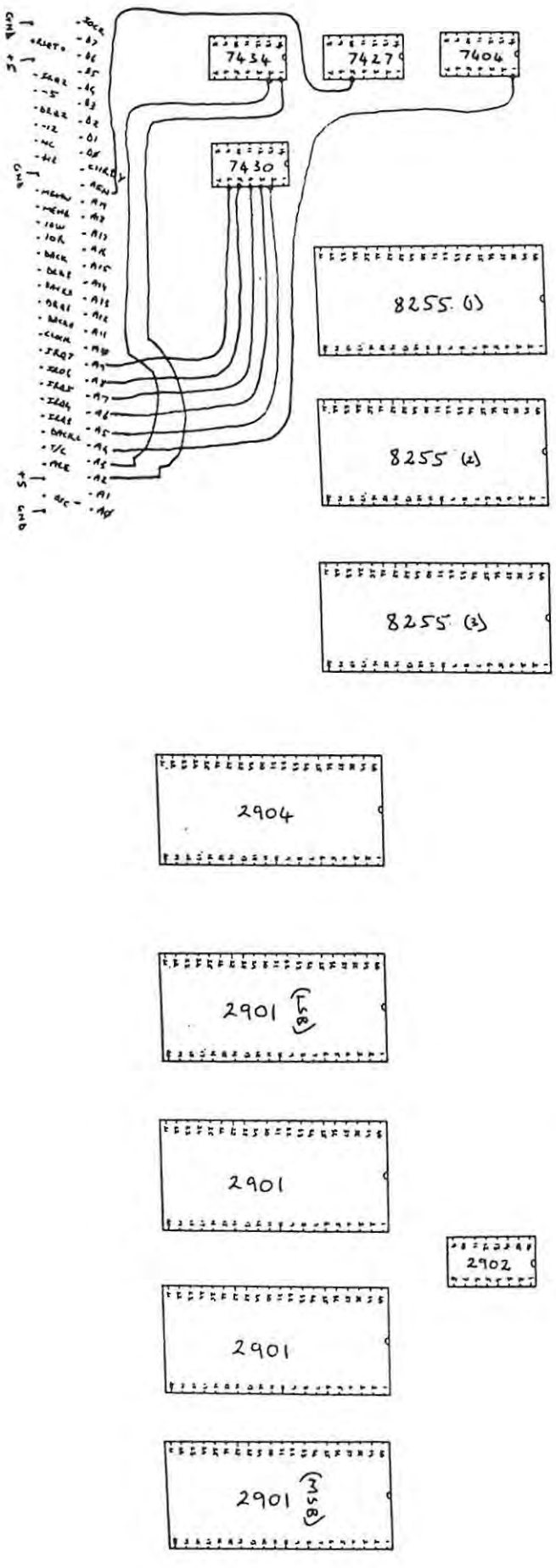




Wire Wrapped Side

Circuit Section: Power Supply (V_{cc})

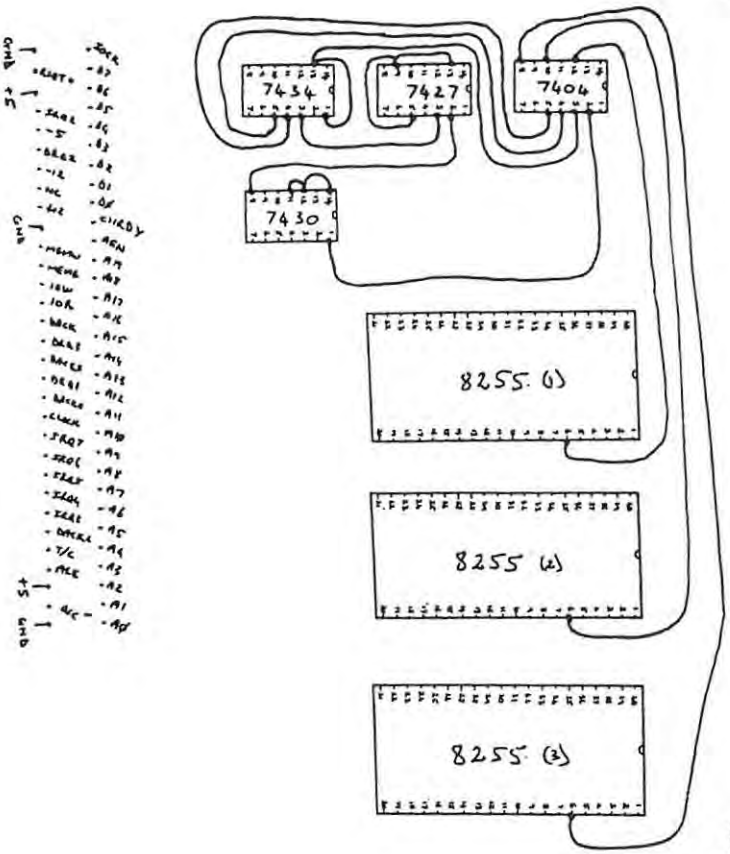
Wire Colour: Red

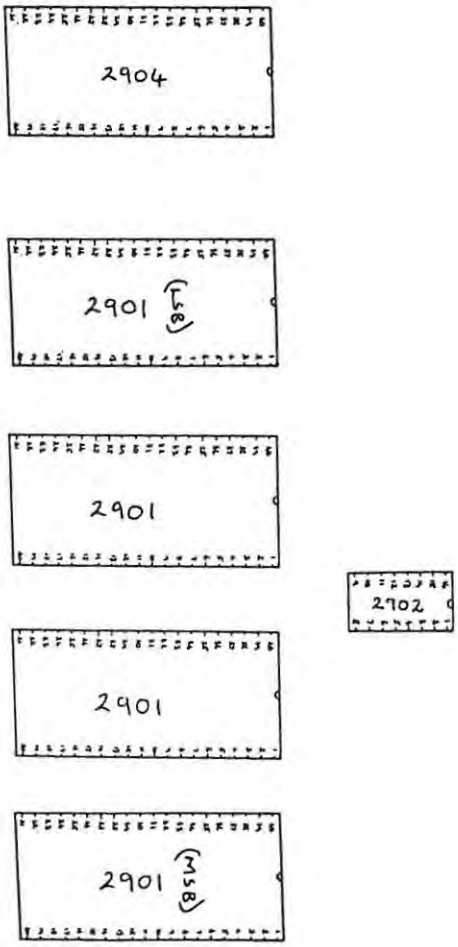


Wire Wrapped Side

Circuit Section: Decode Circuit

Wire Colour: Blue

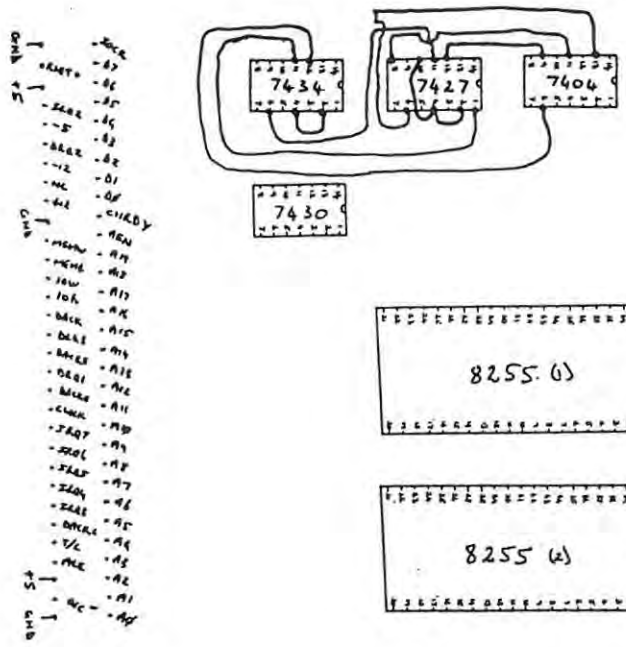




Wire Wrapped Side

Circuit Section: Decode Circuit

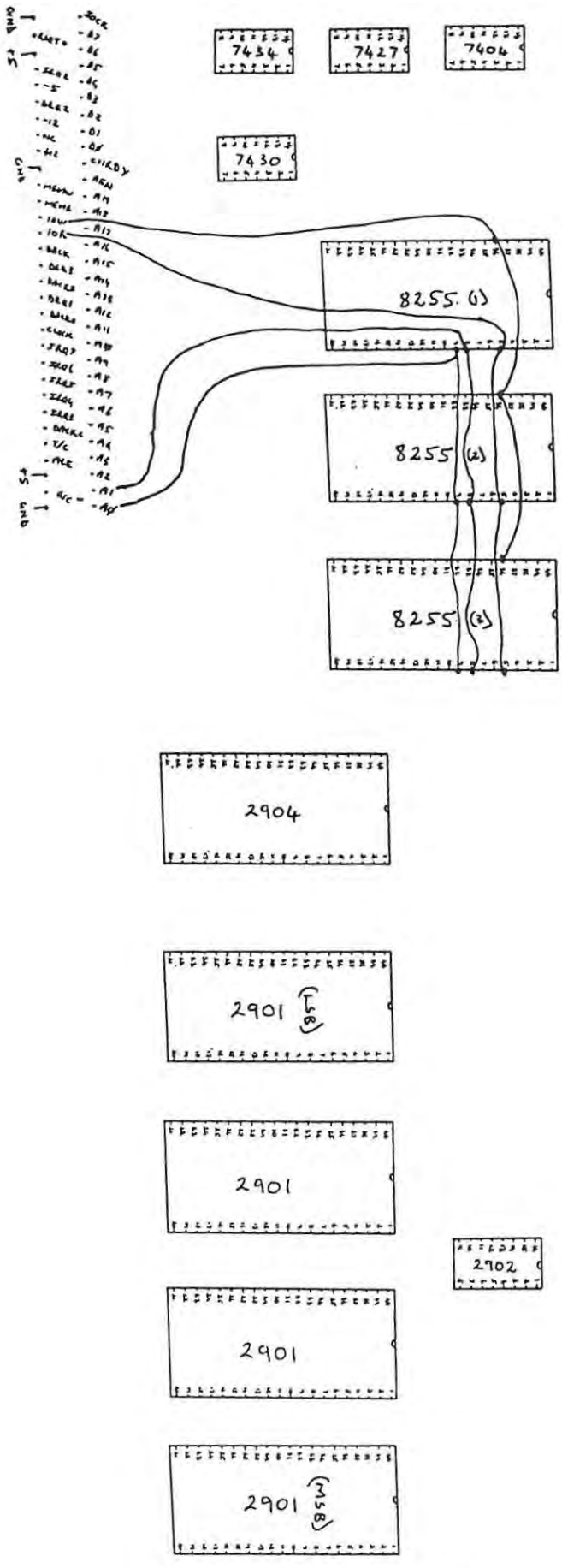
Wire Colour: Yellow



Wire Wrapped Side

Circuit Section: Decode Circuit

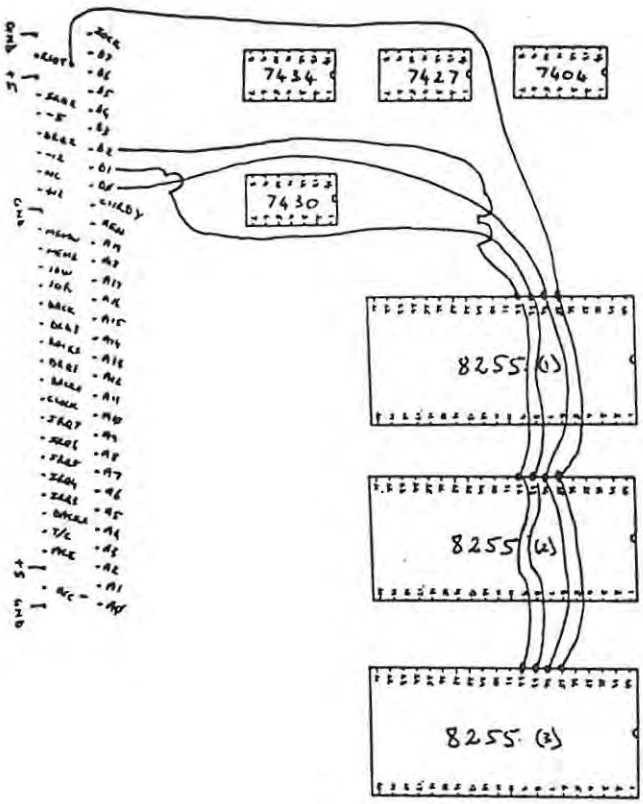
Wire Colour: Yellow

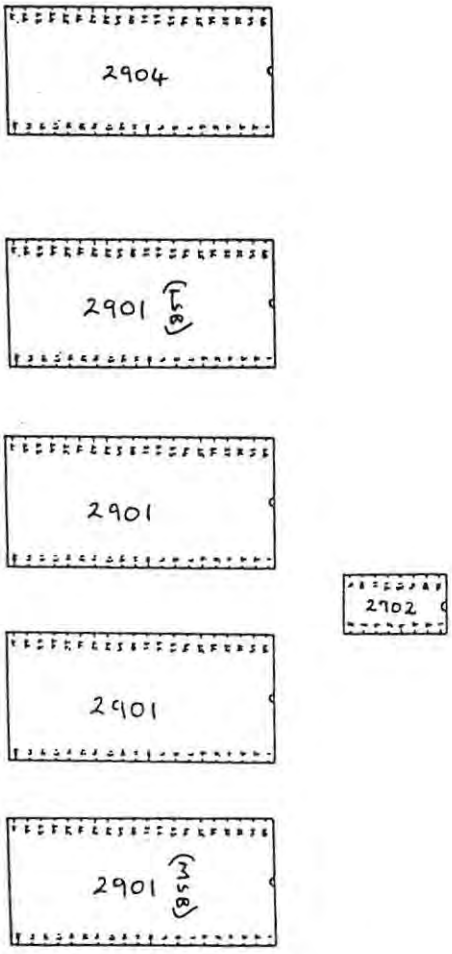


Wire Wrapped Side

Circuit Section: 8255 (1)

Wire Colour: White

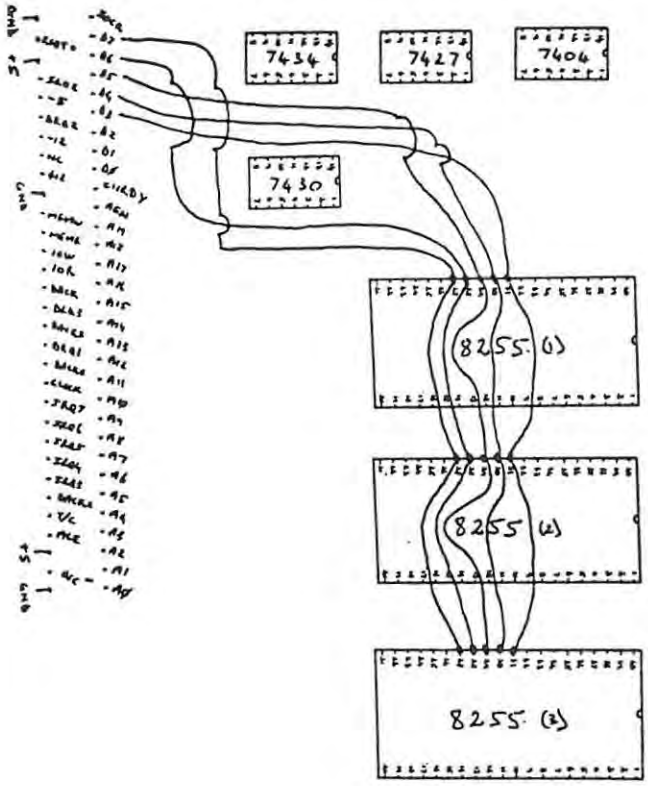




Wire Wrapped Side

Circuit Section: 8255 (1)

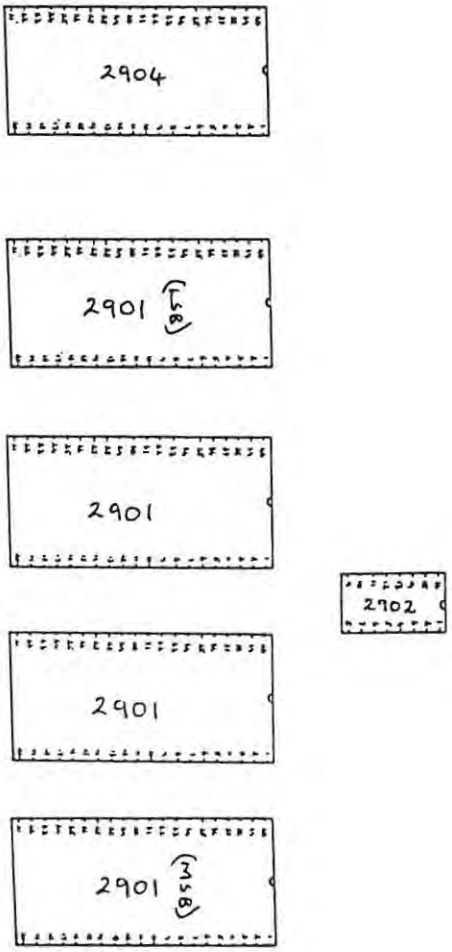
Wire Colour: White

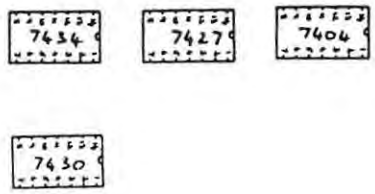


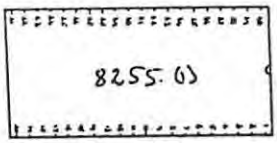
Wire Wrapped Side

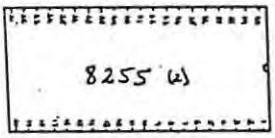
Circuit Section: 8255 lines

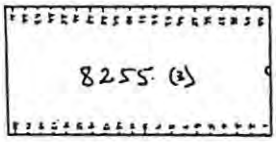
Wire Colour: White







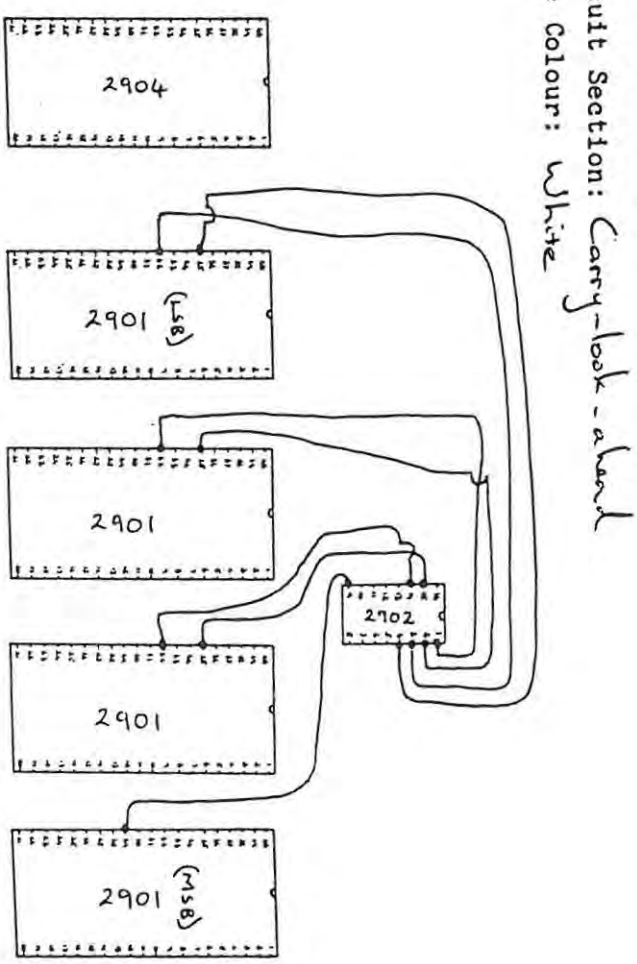


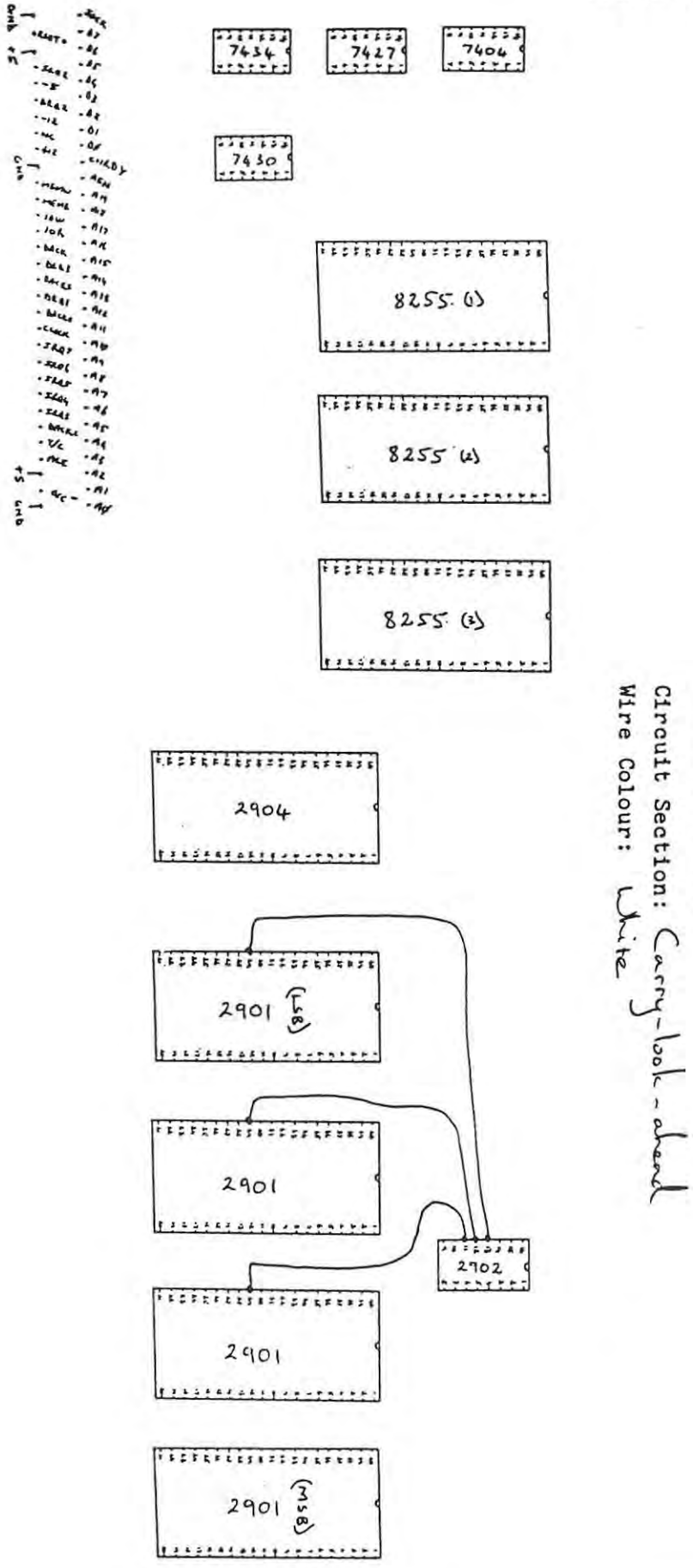


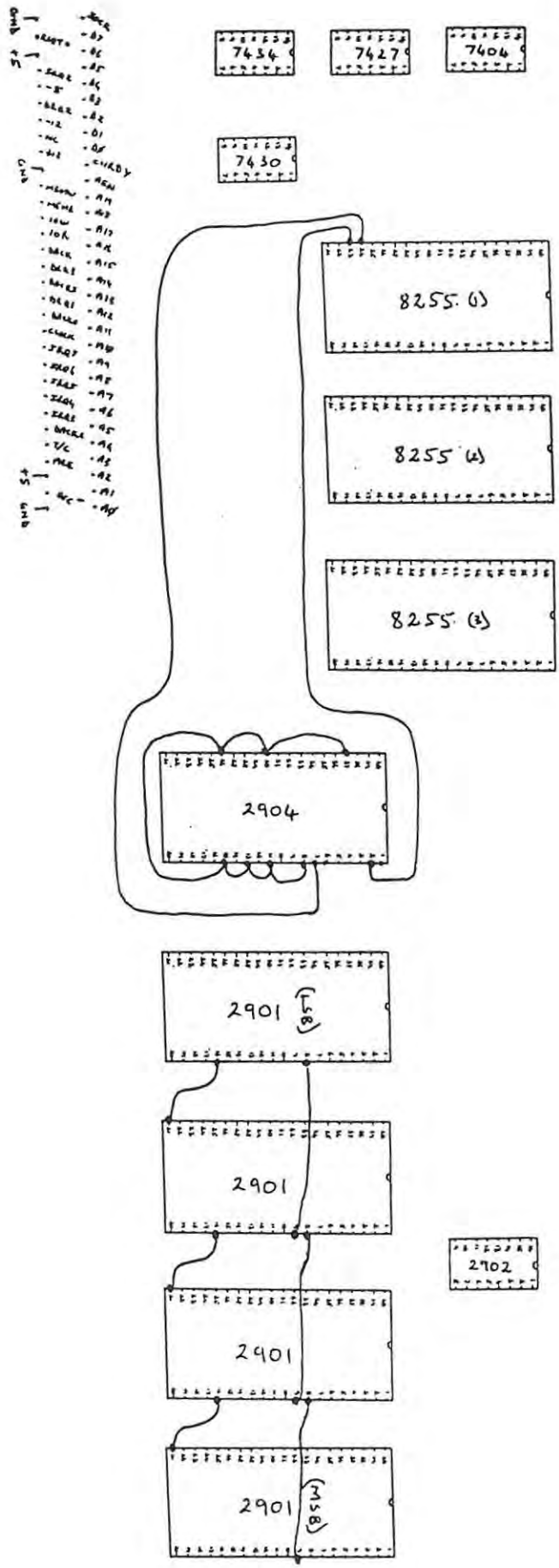
Wire Wrapped Side

Circuit Section: Carry-look-ahead

Wire Colour: White



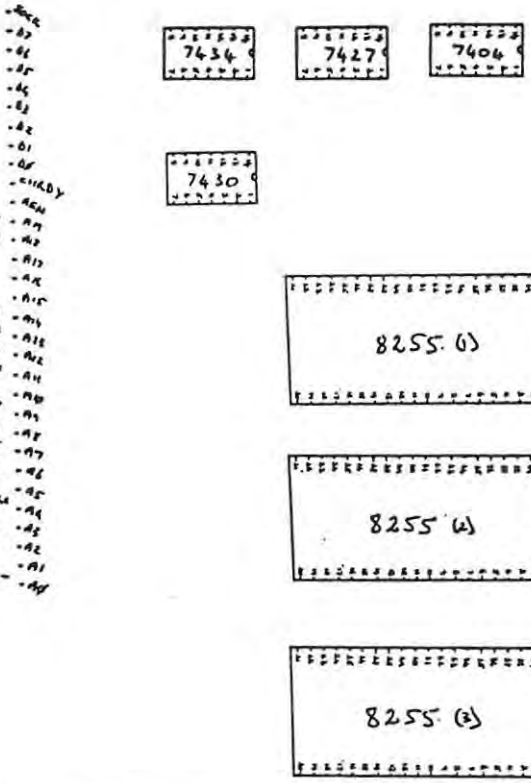

7434
7427
7404
7430
8255 (1)
8255 (2)
8255 (3)
2904
2901 (LSB)
2901
2902
2901
2901 (MSB)
Wire Wrapped Side
Circuit Section: Carry-look-ahead
Wire Colour: White

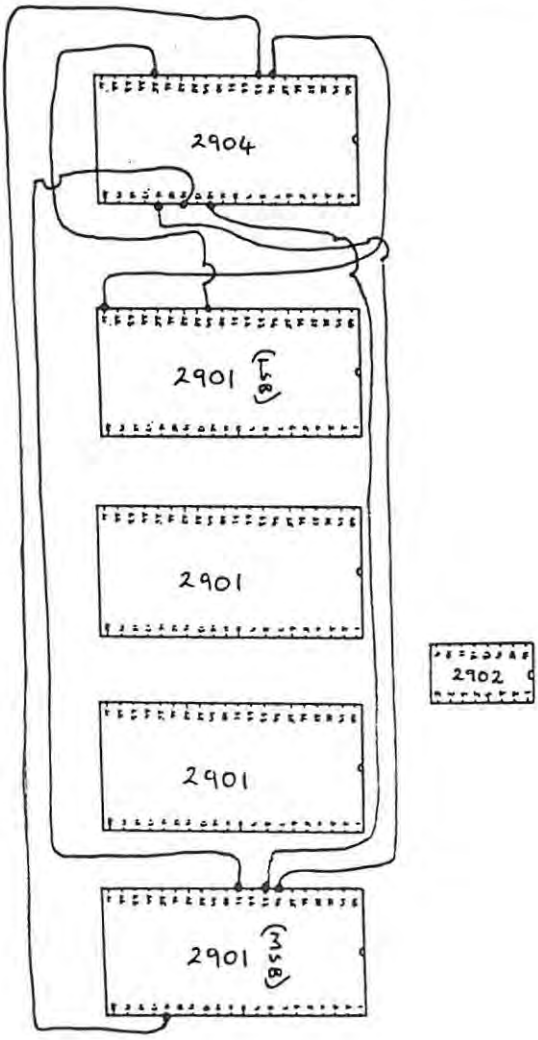


Wire Wrapped Side

Circuit Section: 2904 lines + shift links

Wire Colour: Yellow

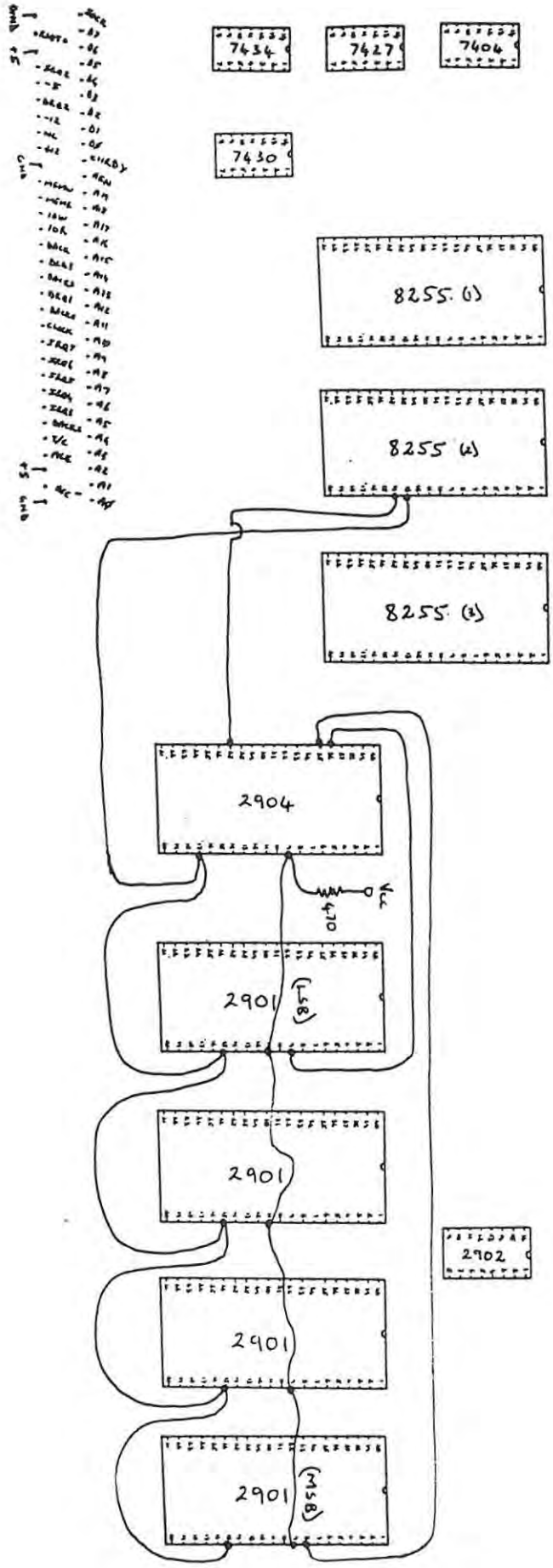




Wire Wrapped Side

Circuit Section: 2904 lines + shift links

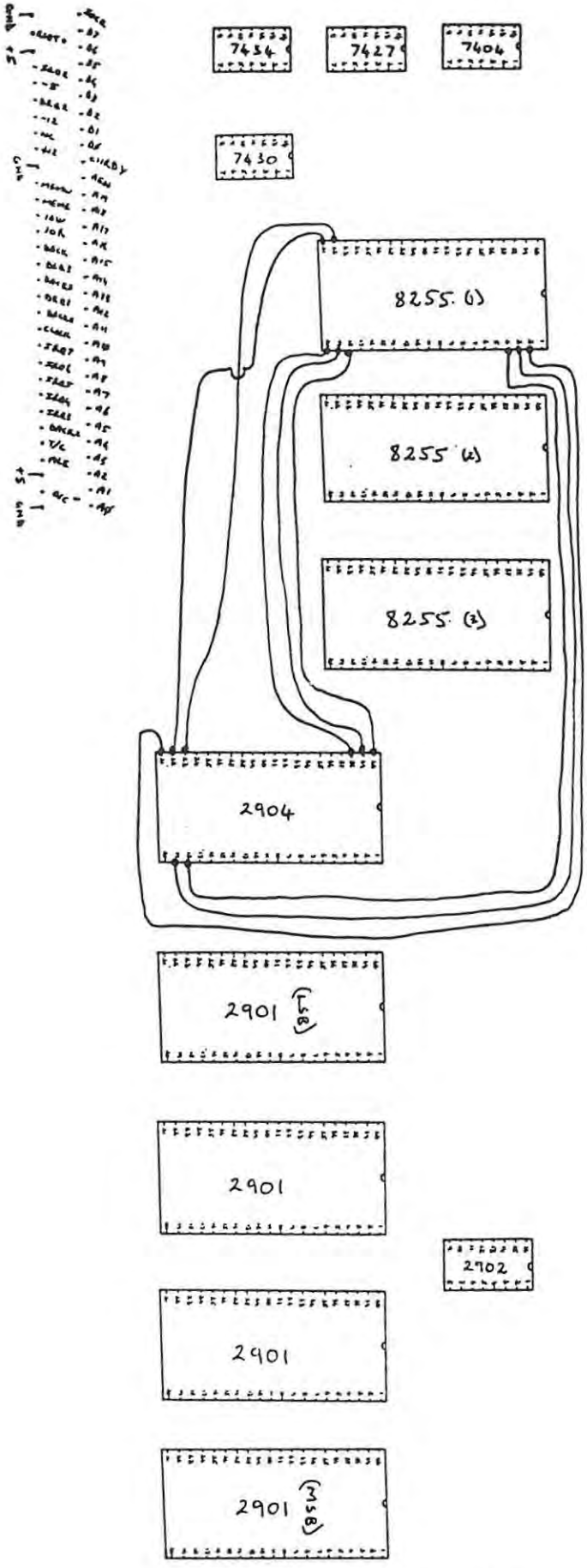
Wire Colour: Yellow



Wire Wrapped Side

Circuit Section: 2904 lines + shift links

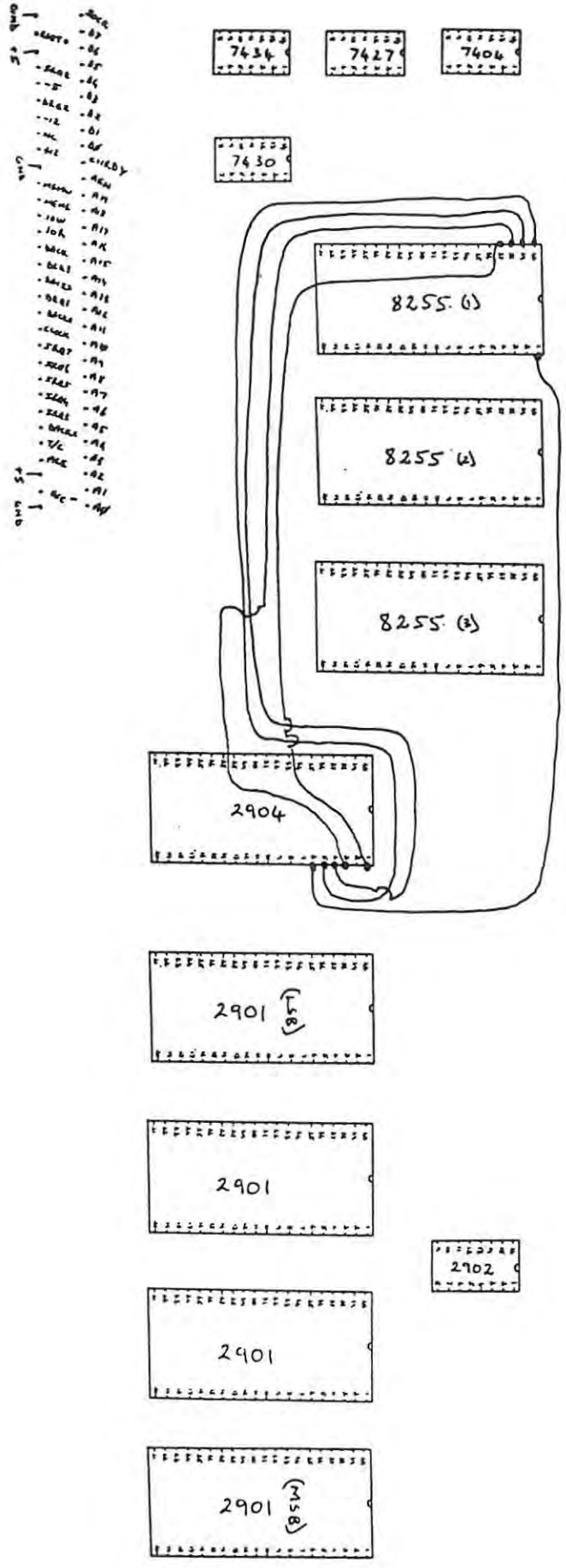
Wire Colour: Yellow



Wire Wrapped Side

Circuit Section: 2904 Control Lines

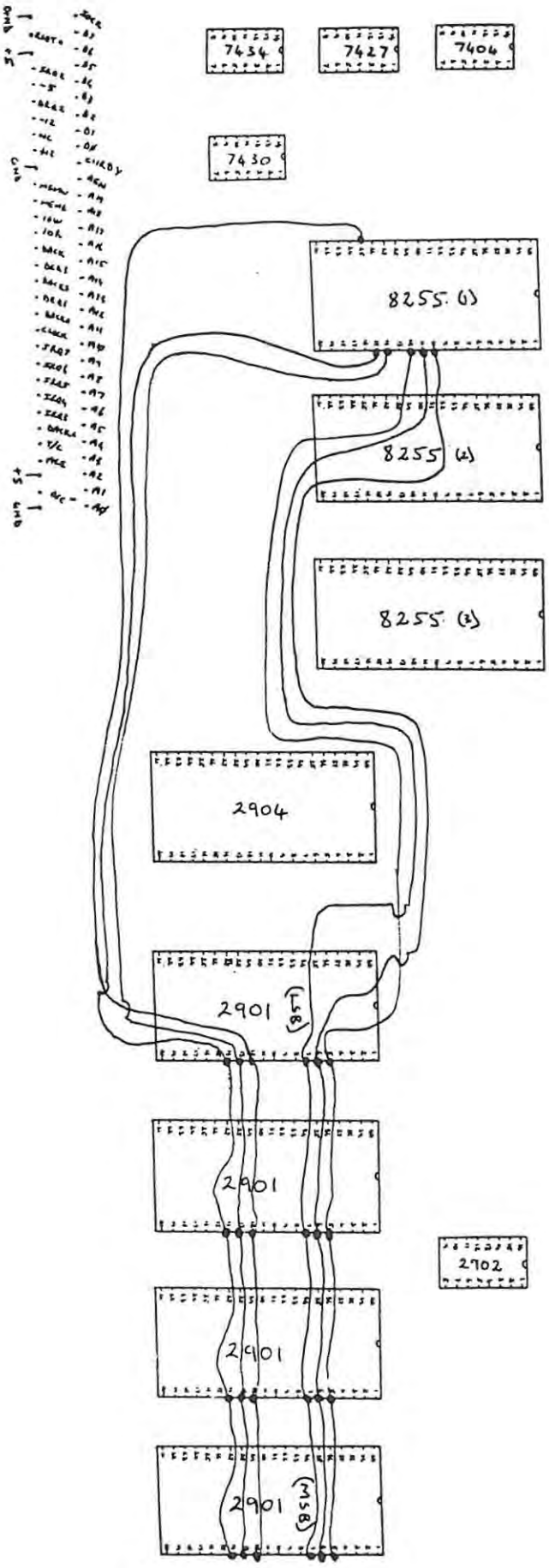
Wire Colour: Blue



Wire Wrapped Side

Circuit Section: 2904 Control Lines

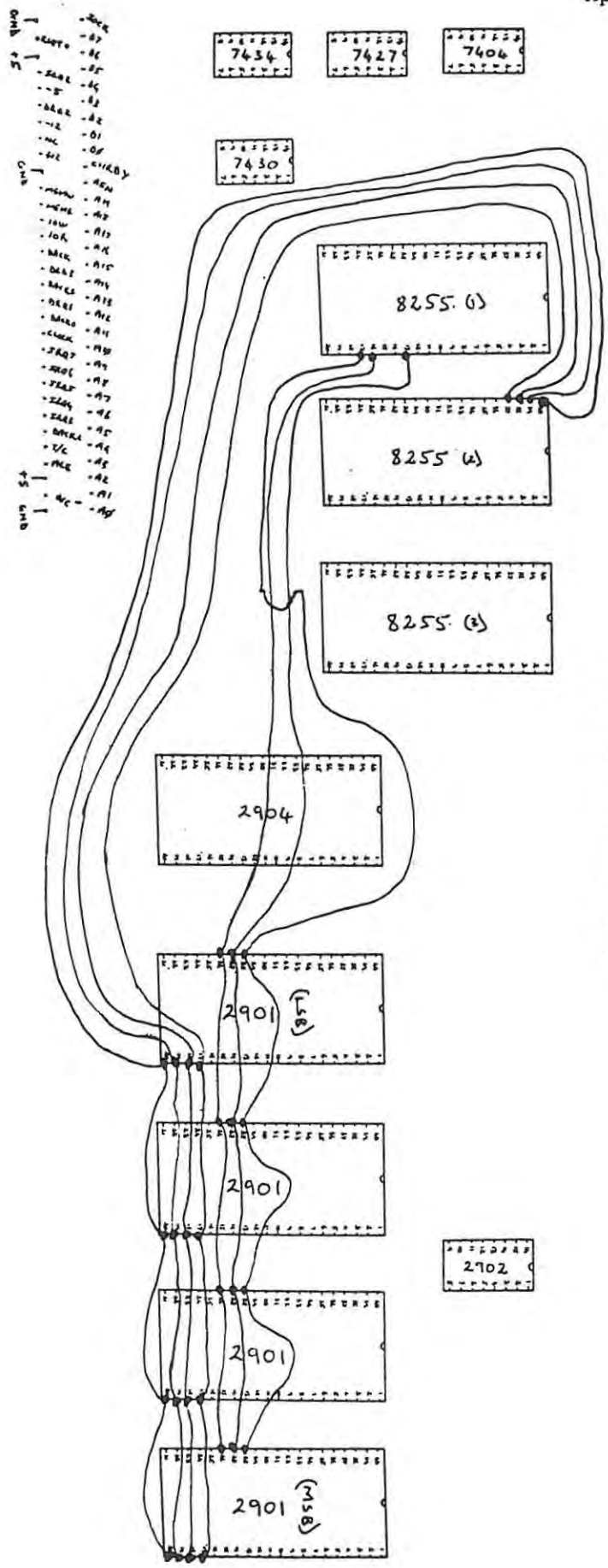
Wire Colour: Blue



Wire Wrapped Side

Circuit Section: 2901 Control Lines

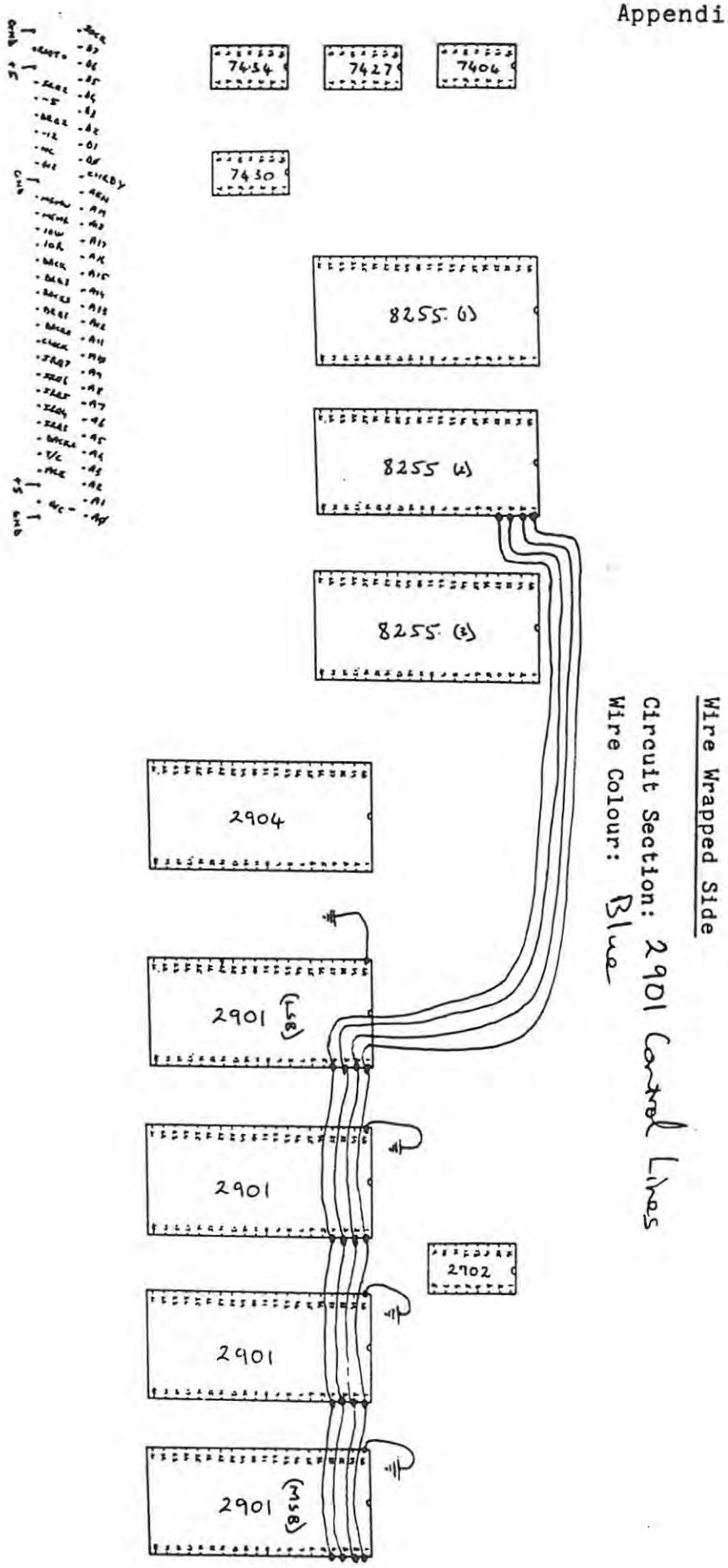
Wire Colour: Black



Wire Wrapped Side

Circuit Section: 2901 Control Lines

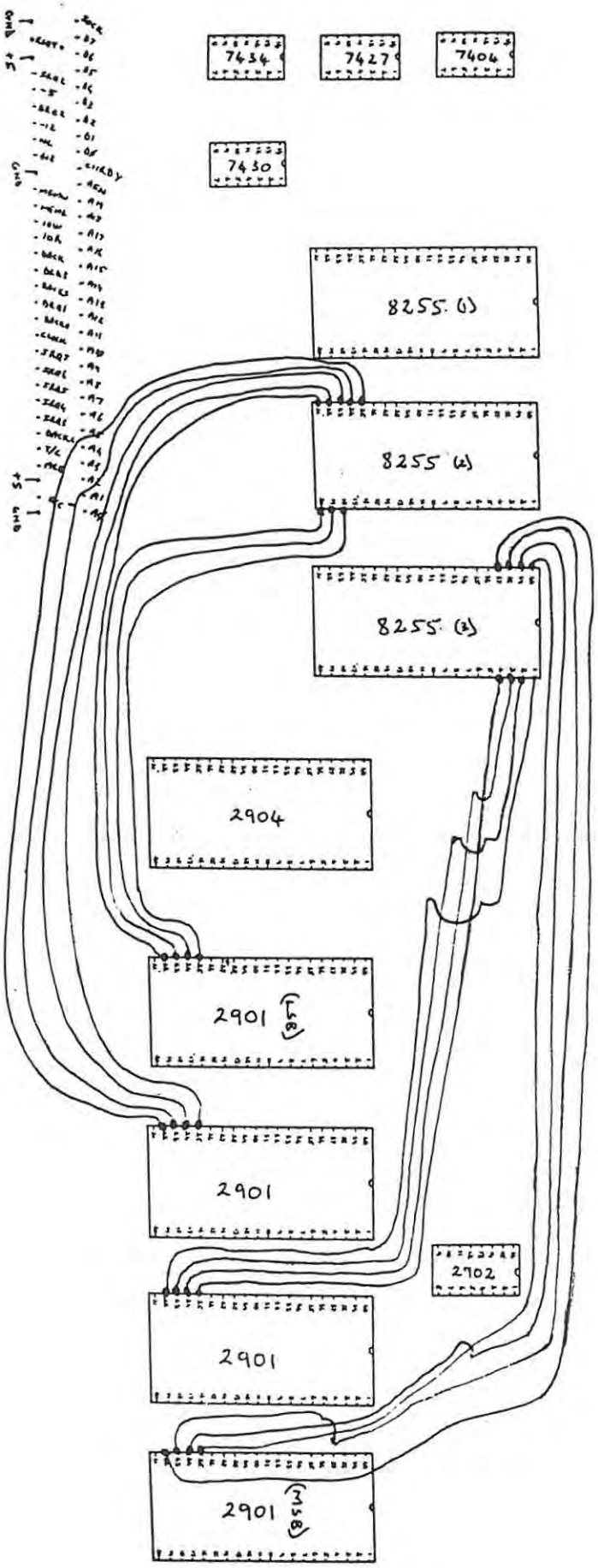
Wire Colour: Black

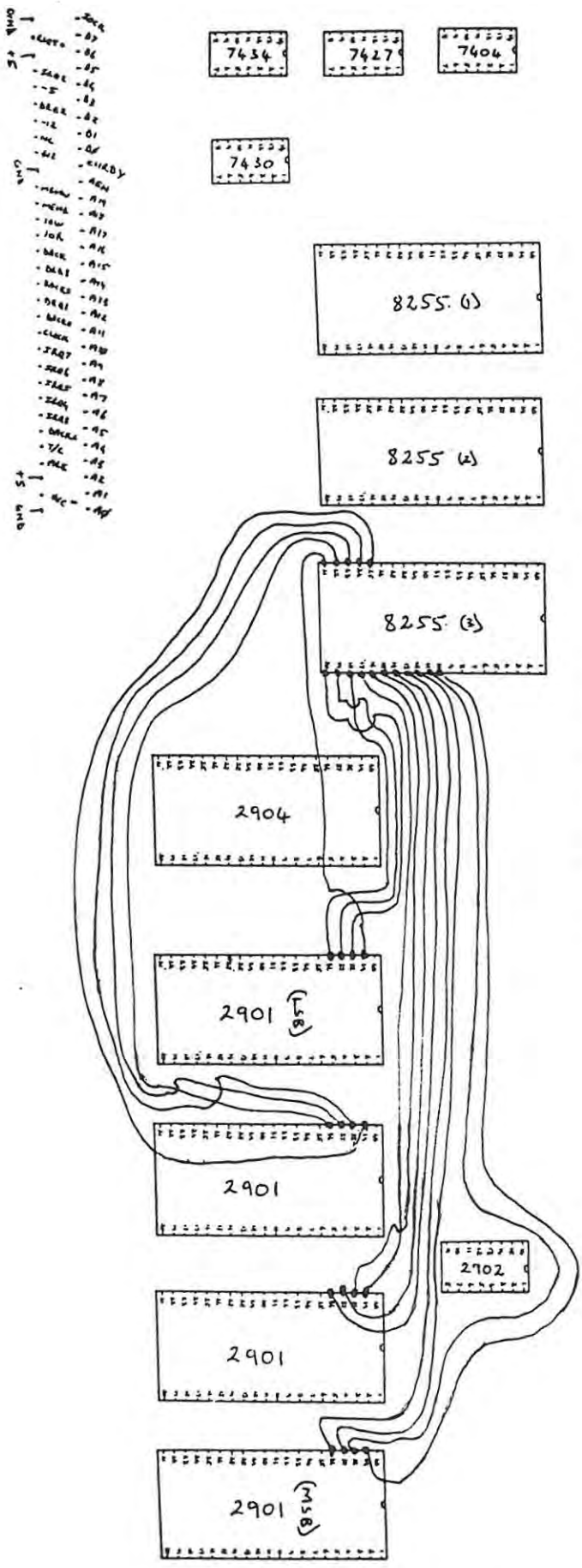


Wire Wrapped Side

Circuit Section: 2901 Control Lines

Wire Colour: Blue


7434
7427
7404
7430
8255 (1)
8255 (2)
8255 (3)
2904
2901 (LSB)
2901
2902
2901
2901 (MSB)
Wire Wrapped Side
Circuit Section: D-bus
Wire Colour: Red



Wire Wrapped Side

Circuit Section: Y-bus

Wire Colour: Red