ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

AMP, A Dynamic Microprocessor

by

Robert Barr, Jerome Becker,
William Lidinsky, and Vernon Tantillo

Applied Mathematics Division

June 1973

NOTICE

This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Atomic Energy Commission, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately owned rights

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency Thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

TABLE OF CONTENTS

	<u>Page</u>
GLOSSARY.	5
ABSTRACT.	7
1. INTRODUCTION.	7
2. FRAME OF REFERENCE.	8
2.1 Microinstruction Word Length	8
2.2 Control Store.	8
2.3 Microcycle	9
2.4 Sequencing	9
2.4.1 Look-ahead Sequencing	11
2.4.2 Current Word Sequencing	11
2.4.3 Probabilistic Sequencing.	11
2.4.4 Modified Current Word Sequencing.	11
3. MACHINE DESCRIPTION	
3.1 Introduction	13
3.2 Memory Organization.	16
3.3 CPU Organization	18
3.4 Microcontrol Word.	22
3.5 Microsequencing.	22
3.6 Timing and Considerations.	24
3.7 Machine Level Instructions	34
3.8 Ease of Hardware Modifications	37
3.9 Manipulators	38
3.10 Operation of Peripherals	39
4. SOFTWARE.	43
4.1 Microassembler	43
4.2 Macroassembler	43
5. EXPERIENCES	
5.1 Machine and Assembly Language.	44
5.2 Microcode Analysis	45
5.2.1 Micro-operations per Microinstruction	46
5.2.2 Microinstruction Bit Utilization.	48
5.2.3 Microprocessor Resource Utilization	48
5.2.4 An Execution Time Comparison.	53
6. CONCLUSION.	57
APPENDIX A: MANIPULATOR FUNCTIONS	
Byte Manipulator Control Bits	59
Unary Manipulator Control Bits.	60
Binary Manipulator Control Bits	61
APPENDIX B: MICROINSTRUCTION SUBFIELD DEFINITIONS.	64
ACKNOWLEDGMENTS	67
REFERENCES.	68

LIST OF FIGURES

<u>No.</u>	<u>Title</u>	<u>Page</u>
1.	Microcycle Timing.	10
2.	Argonne Microprocessor	14
3.	AMP Console.	15
4.	AMP Virtual Memory Map	17
5.	AMP Central Processor.	19
6.	Local Store Addressing	21
7.	Microinstruction Sequence Control Bits	25
8.	Fast Microcycle Timing	26
9.	Slow Microcycle Timing	27
10.	Timing Generator Block Diagram	32
11.	AMP Machine Instruction.	35
12.	Control Memory Organization for Using INR Fields	36
13.	Typical Peripheral Communication Paths	40
14.	Peripheral Timing.	41
15.	Microoperation Histogram	47
16.	Microword Bit Utilization.	49
17.	Bus Utilization.	50
18.	Manipulator Utilization.	51
19.	Local Store Utilization.	52
20.	Jump Address/Immediate Field Utilization	54
21.	Matrix Addition Execution Times.	55

LIST OF TABLES

<u>No.</u>	<u>Title</u>	<u>Page</u>
I.	Microcontrol Word Organization.	23
II.	Fast Cycle Timing Information	29
III.	Slow Cycle Timing Information	30
IV.	CPU Timing Pulse Functions.	31

GLOSSARY

The following definitions are given to minimize ambiguities that might occur in the succeeding text.

Assembly instruction: a mnemonic instruction which is mapped by an assembler into one or more machine instructions.

Machine instruction: a bit pattern which is interpreted by the executing control hardware or microsubroutine.

Microinstruction: a bit pattern normally stored in control memory which controls, at a primitive level, the processor hardware. It specifies the micro-operations that take place during one microcycle. The microinstruction may contain a single operation, as is usual in vertical microprogramming, or specify several concurrent operations (horizontal microinstruction).

Microprogram or microsubroutine: a sequence of microinstructions. A machine level instruction is normally executed by a microsubroutine.

Microcycle: the time necessary to execute one microinstruction.

Micro-operation: a primitive microprogram action (e.g., addition, shift, transfer, branch, etc.).

Microprocedure: a sequence of micro-operations, not necessarily in contiguous microinstructions.

Virtual memory: memory space as seen by the microprocessor. Virtual memory may actually consist of a variety of memory types and peripheral devices.

Control memory: the high speed memory in which microinstructions are stored.

Local store: a small, very high-speed memory located within the CPU that operates synchronously with the CPU. It is the fastest of all memory types.

Manipulators (or ALU's): entities within the processor which perform manipulations of the data under direction of the microinstruction (e.g., add, shift, logical AND, byte packing, etc.).

Busses: multiple-source/multiple-destination paths by which data may be routed between registers, manipulators, memories, and I/O devices.

AMP, A Dynamic Microprocessor

by

Robert Barr, Jerome Becker, William Lidinsky, and Vernon Tantillo

ABSTRACT

A horizontally structured microprogrammable processor (AMP) designed as a tool for microcontrol, language, and processor design research, is described. The machine employs a minimally encoded microcontrol word, a general multiple-bus structure, high speed local storage, several arithmetic/logic units, and completely asynchronous memory referencing. These features combine to yield substantial low level parallelism and a very versatile microinstruction. In addition, hardware modification is easily accomplished, further supporting the experimental nature of the processor.

The paper first establishes a frame of reference for both microcontrol and microprogramming; then the processor organization and implementation is described. Next a brief description of the software is provided. Finally, initial work with the machine is discussed and related to the design of microcontrolled processors and associated languages.

1. INTRODUCTION

An important trend in the design of computing systems is the implementation of the CPU as a horizontal microcontrolled processor containing several ALU's, local stores, busses, etc. Microcontrol offers a variety of design and programming advantages such as machine level compatibility between computers, variable architecture and organization, and the ability to program at a level much closer to the actual hardware. In addition, if the control store is easily modified, systems and languages can be tailored to the application at hand and microprograms can be dynamically controlled. The resources and the horizontal structure of the controller provide the advantage of low level parallelism.

The proper organization of these machines, the effective use of parallelism, instruction stream optimizing, implications of dynamically alterable microstores, and the direct implementation or support of higher level languages in microcode are but a few of the unanswered questions of microprocessing.

2. FRAME OF REFERENCE

A substantial amount of literature exists in the area of microstructure control logic [2] and microprogrammable processors. In describing a microprogrammable machine, it is desirable to establish a frame of reference with respect to the literature and to resolve ambiguities.

2.1 Microinstruction Word Length

One method of classifying microstructure control is by control-word length. This classification represents the two ends of a continuum of word lengths. The "short" bit length of a "vertical" microinstruction is generally from 10 to 30 bits. Because of the limited number of bits, the organization of the vertical control word reflects a highly encoded approach in order that the word may contain as much information as possible. This approach leads to a loss of generality within the microinstruction and it slows execution by the amount of time required to decode the instruction. Indeed, vertical microinstructions are quite similar to conventional machine instructions and generally include complex basic timing cycles.

"Horizontal" or "long" microinstructions (i.e., 60 bits or more) are minimally encoded. Small fields within the instruction are used to control CPU resources such as ALU's, busses, registers, etc. This type of microinstruction is much more general than the vertical type. Because of the minimal encoding, very little decoding is required and execution is more rapid. Also, the large word size provides the control capability for the execution of several basic processes or "micro-operations" for each microinstruction. However, in order to efficiently do this, a very close interrelationship between microinstructions must exist.

A few current machines [3] have control store widths of several hundred bits; but, in these cases each control-word contains several smaller microinstructions.

2.2 Control Store

Inherent to the concept of microcontrol is the high-speed writable control memory [4]. The speed of the micromemory should approximate the logical operational speed of the processor. Writability introduces the

concept of "dynamic" microprogramming where the microcode can be changed to provide more tailored support for the application at hand, or where the microcode can modify itself [5].

2.3 Microcycle

Redfield [6] classifies microcycles as monophase or polyphase. In the case of a monophase microcycle, a microinstruction generates controls that are used during only one clock pulse. There is in effect one execution clock pulse per microcycle.

Conversely, in the polyphase approach there are several clock pulses per microcycle. One can further break the polyphase microcycle into "simple polyphase" and "complex polyphase." An examination of Figure 1 shows that in the simple polyphase approach there is a single set of clock signals during the time that the microinstruction is stable. In the complex polyphase approach, a set of timing pulses is repeated several times.

Both the strict monophase and the complex polyphase microcycles tend to be impractical. The former, due to the large number of microinstructions (and thus control memory references) per microsubroutine, and the latter because it fails to efficiently use the microinstruction.

2.4 Sequencing

Sequencing refers to the determination of the chronological order in which microinstructions are to be executed.

The control memory which contains the microinstructions has a finite access time. Since the microinstruction is changing during this access time, it cannot be executed. Consequently, it is desirable to overlap the execution time of the current microinstruction with the access time associated with the fetch of the next microinstruction. No problem exists as long as the next microinstruction address is not dependent upon the results of the execution of the current microinstruction. However, overlapping cannot occur if a conditional branch is required which depends on the results of the current microinstruction.

Microsequencing schemes can range from complete overlap to none at all.

Fig. 1. Microcycle Timing

2.4.1. Look-ahead Sequencing. Restricting the determination of the next microinstruction to the results of some previous one other than the current microinstruction is one approach that allows overlapping. The IBM S/360, Model 50 CPU is an example of a machine which uses this approach [8]. This approach puts a burden on the microprogrammer. In order to take advantage of the complete overlap, the microprogrammer must do branch testing and set up branch conditions in one word, do something useful in the next word (which occurs before the actual branching), and then branch. If something useful cannot be done, a microinstruction has to be wasted to allow the conditional branch to occur. While clever microprogramming can make use of this additional instruction, it is not always possible. When it is not possible, both microinstruction time and memory space allocated for the microinstruction are lost. The upper bound on this lost space and time is equal to the number of conditional microbranches.

2.4.2. Current Word Sequencing. In this approach the current microinstruction determines the address of the next microinstruction. Here no overlapping can occur. Enough time must be allowed in the microcycle to complete the execution of the microinstruction before the address of the next microinstruction is determined. This approach has the advantage of easy microprogramming and the disadvantage of wasted access time.

2.4.3. Probabilistic Sequencing. Probabilistic sequencing involves fetching the most likely alternative microinstruction on an overlapped basis, backing up and refetching if necessary. This scheme does not involve microprogramming difficulties and statistically it could fare as well as the look-ahead scheme. One has to compare the probability of being able to successfully use the last microinstruction before the branch in the look-ahead approach against the probability that the "most likely alternative" is correct in this approach. In addition, this scheme appears to be the most complex to implement.

2.4.4. Modified Current Word Sequencing. Another approach that is not too difficult to implement involves having two different cycle times; a short cycle for unconditional branching where overlapping access and execution times occur, and a long cycle where control memory access time occurs

after execution time. This scheme, like the previous one, has the advantage of not requiring the microprogrammer or microassembler to "look-ahead." However, it must perform a long cycle for each conditional branch. In this approach lost execution time is equal to the upper bound of the lost time under the look-ahead approach; but, microcontrol word space is not lost.

3. MACHINE DESCRIPTION

3.1 Introduction

AMP is a horizontally structured, simple polyphase microprocessor with the capability of performing several micro-operations per microinstruction. It has a 2048-word monolithic control store and uses a modified version of current word sequencing with two different microcycle times. The microinstruction length is 74 bits. All memory and peripherals are organized as parts of a single virtual memory. The CPU consists of a 64-word local store, three ALU's, and a variety of registers.

This section describes the microprocessor. The AMP microcycle spans a sequence of ten timing signals. This fact, coupled with the parallelism of the CPU and the large microword, yields versatile microinstruction. For example, a single microinstruction can fetch up to three operands, perform two data manipulations, store two results, perform a conditional microsequencing operation based on the results of manipulations, and execute a memory reference. This is in contrast to machines such as the one proposed by Cook and Flynn [5].

The microprocessor is characterized by a multiple high-performance bus structure with several sources and destinations for each bus. This differs from several other existing or proposed machines with multiple bus structures [3,7].

The 74-bit microword which controls the processor uses judicious but minimal encoding to extend its capabilities. Microsequencing is done on a current word basis rather than on a look-ahead basis. This eases the task of the microprogrammer. Three data manipulators and an interleaved 64-word local store function simultaneously.

The microprocessor and its control console are shown in Figs. 2 and 3. Because much of the microprocessor is composed of high-speed TTL and Schottky logic, special precautions were taken during physical construction to reduce the noise generated by very fast pulse edges. These precautions include the use of a continuous ground plane between all integrated circuit boards, and transmission line termination techniques for all backboard wiring. The

Fig. 2. Argonne Microprocessor.
ANL Neg. No. 145-1750.

Fig. 3. AMP Console. ANL Neg. No. 145-1748.

control console provides indicators for the simultaneous display of any six of a possible 24 registers. Control memory may also be loaded from the console.

3.2 Memory Organization

As Cook and Flynn [5] point out, all processors have a hierarchy of memories, going from the fastest (e.g., registers) to the slowest (e.g., tapes). AMP has four levels which are: (1) 64 local general-purpose registers of 16 bits each; (2) the control memory which consists of 2K of 96-bit words of which 74 bits are presently used; (3) 16K of 750 ns core of 16-bit words; and (4) 8K of 1.75 microsecond core of 16 bits. The 64 general-purpose registers along with 12 registers for special use are integral to the CPU itself. The other levels of memory are contained within a single virtual memory as shown in Fig. 4. The virtual memory also contains an area reserved for peripherals. Thus, every peripheral becomes simply one or more addresses in virtual memory and any memory reference action may address any of the levels of memory or the peripherals with equal facility. In a sense, all levels of memory beyond the general-purpose registers may be considered peripheral to the processor.

The 64-register local store is normally used for the temporary storage of operands which are either generated by the CPU or obtained from virtual memory. All CPU manipulations are performed on the contents of the local storage registers or the twelve special-purpose registers.

The control memory contains the microinstructions and the constants that are necessary to the execution of the microcode. It may also be used as general-purpose memory; however, its size limits its use for this purpose. The control memory is organized as 2048 words of 96-bit lengths. However, it is addressed in virtual memory as 12288 sixteen-bit words. The control memory may be expanded up to 2048 words $\times$ 128 bits. Depending on the manner of access, the control memory is organized so that it is compatible with its microcontrol requirements as well as with virtual memory. The availability of the control memory to the CPU provides the necessary features for dynamic microcontrol.

Fig. 4. AMP Virtual Memory Map

The core portions of virtual memory are normally used for machine level programs and data. The 1.75 microsecond core is common to a PDP-7 computer which acts as an I/O channel for the system [1]. The 750 ns core is directly accessible from a disc and a graphic vector generator.

Communication between the processor and virtual memory is fully buffered and completely asynchronous, and the various portions of the virtual memory are completely independent of one another. Consequently, the microprocessor can cycle these various memory segments simultaneously, which improves execution time. Caution must be used, however, and a detailed knowledge of timing constraints is necessary.

Since memory references are asynchronous, additional memory can be added to the unused area with ease. The microprocessor is capable of accepting memories of any access or cycle time. Bus and internal timing propagation times are such that approximately 35 ns is added to the effective cycle time of any memory.

3.3 CPU Organization

The microprocessor CPU is shown in Fig. 5. All busses labeled B1 through B5, and BA, BO, and BI are 16 bits wide. Each internal bus (i.e., B1-B5) provides eight data transfer paths. Starting at the upper left of the figure, the box labeled R0 provides a source of zeros to all internal busses. The byte manipulator performs operations on 8-bit bytes; such as swap the left and right 8-bit bytes, or unpack two bytes. The unary manipulator provides up to 32 operations on one operand, such as increment, decrement, shift left or right, or rotate. The binary manipulator provides up to 32 operations on two operands; examples are add, subtract, AND, OR, etc. RA and RB are general-purpose registers. RSW is a switch register which is available to the operator.

There are two local stores or scratch pads labeled SPE and SPO. Each scratch pad is composed of thirty-two 16-bit words and each scratch pad is addressable either via bits in the microcontrol word or through a register labeled SPAR. The addressing scheme is such that three locations per microcycle may be specified. For example, SPE and SPO may be addressed via the microcontrol word as source registers, and addressed via SPAR as a destination register. The local store addressing scheme is shown in Fig. 6. If

Fig. 5. AMP Central Processor

the scratch pads are addressed from USDR bits, then there are two separate local stores of 32 words each. If the scratch pads are addressed from SPAR, the result is one local store of 64 words. SPAR may be incremented so that it may be used as a source and destination address within one microcycle. The high order bit of SPAR can be set only from bus 3. If SPAR is incremented wraparound of SPAR addresses 0 through 31 or wraparound of SPAR addresses 32 through 63 occurs. The access time for all local stores is 60 ns.

There are three input/output registers: BMAR, MDRO, and MDRI. These three registers are necessary to the asynchronous I/O and memory reference operations.

The general flow of information in Fig. 5 is in the clockwise direction starting with a source register such as RA, RB, or a scratch pad, through one of the lower busses (B3, B4, or B5), then one of the ALU's, an upper bus (B1 or B2), and finally to one or more destination registers (such as RA, RB, or a scratch pad). The configuration of the internal busses (B1 through B5) is such that a binary operation and a unary operation may take place simultaneously.

Referring to the lower portion of Fig. 5, the box labeled UMAR (micro-memory address register) has six sources. The O and R fields of INR transferred into the UMAR provide an automatic branch table. The microstack provides for hardware microsubroutine linkage. (The microstack at this time contains four words but is easily expandable to any length.) A fourth source for the UMAR is 16 bits of the microstorage data register. This path yields a jump address capability. The PDP-7 source for the UMAR is used while loading the control memory from the PDP-7. The sixth input to the UMAR is the BA, or address bus. This particular path is of significant interest because it allows the microprogrammer to dynamically change the contents of the control memory. The UMAR may be incremented by one for normal program sequencing, or incremented by two for a microskip operation.

The bit patterns contained within USDR control the data flow and register sets throughout the microprocessor CPU. The USDR is 96 bits wide and expandable to 128. At present 74 bits are used. (Refer to Section 3.4, Microcontrol Word, for details on the USDR.) The microsequence control can

SPE		SPO	
μ BIT ADDRESS	SPAR ADDRESS	μ BIT ADDRESS	SPAR ADDRESS
00	00	00	01
01	02	01	03
02	04	02	05
03	06	03	07
04	08	04	09
05	10	05	11
06	12	06	13
07	14	07	15
08	16	08	17
09	18	08	19
10	20	10	21
11	22	11	23
12	24	12	25
13	26	13	27
14	28	14	29
15	30	15	31
16	32	16	33
17	34	17	35
18	36	18	37
19	38	19	39
20	40	20	41
21	42	21	43
22	44	22	45
23	46	23	47
24	48	24	49
25	50	25	51
26	52	26	53
27	54	27	55
28	56	28	57
29	58	29	59
30	60	30	61
31	62	31	63

Fig. 6. Local Store Addressing

receive inputs from bus 1 or bus 2 and test for conditions on these busses. Also, it can receive jump address information directly from the microstorage data register. A conventional priority interrupt system is used in the microprocessor. The IPR (interrupt pending register) can contain up to eight external interrupts which are then logically ANDed with the IMR or interrupt mask register. The results of this logical AND are fed to the Priority Logic and the Address Encoder which provides a unique hardware address for each interrupt. This hardware address can be made use of at the discretion of the microprogrammer for servicing interrupts.

3.4 Microcontrol Word

The microcontrol word organization shown in Table I has been minimally encoded so that most of the CPU resources are available during every microcycle. Encoding has been done on bits that control resources wherein exclusivity is a fact. For instance, the internal busses B1 through B5 are each controlled by three bits which are fully encoded to provide eight sources to each internal bus. In order to conserve microcontrol word bits, a mode bit is used which determines the use of bits 32 through 40. Microcontrol word bits 48 through 52 also have a dual role designed to minimize control word size. These bits are used for direct addressing of SPO or for an extension of the immediate field (which is normally 11 bits). If this field is not explicitly used by the microprogrammer for SPO addressing, it is automatically used as an extension of the immediate field. With the exception of the above two double usage cases, all fields of the microcontrol word are available for allocation of resources during each microcontrol cycle.

3.5 Microsequencing

As previously mentioned, AMP uses current word sequencing with two different microcycle times. Bits 64-73 of the microinstruction control sequencing within the CPU; bits 1-11 are used as the microjump address when required. Bits 64-70 actually determine the location of the next microinstruction. Bits 71-73 provide additional degrees of freedom to the sequencing operation.

Table I. MICROCONTROL WORD ORGANIZATION

Function	Field	Function	Field
MODE	0	UNARY MANIP. CONTROL	41-45
JUMP ADDR. OR IMMEDIATE FIELD	1-11	SPAR CONTROL	46-47
RA INPUT CONTROL	12-13	IMMEDIATE FIELD EXT. OR SPØ ADDR.	48-52
BMAR INPUT CONTROL	14-15	SPØ INPUT CONTROL	53-54
B1 INPUT CONTROL	16-18	SPE ADDR.	55-59
B2 INPUT CONTROL	19-21	SPE INPUT CONTROL	60-61
B3 INPUT CONTROL	22-24	LOCAL STORE ADDR. SOURCE	62-63
B4 INPUT CONTROL	25-27	JUMP CONTROL	64-66
B5 INPUT CONTROL	28-30	RETURN (POP STACK)	67
MDRØ INPUT CONTROL	31	SKIP	68-70
[RB INPUT CONTROL	32-33	STORE RTN. ADDR. (PUSH STACK)	71
* INR INPUT CONTROL	34	UPPER BUS SELECT FOR SEQUENCE TESTING	72
[BINARY MANIP. CONTROL	35-40	CONTINUE	73
[I/O CONTROL	32-34		
IMR INPUT CONTROL	35		
* UMAR INPUT CONTROL	36-37		
[BYTE MANIP. CONTROL	38-40		

* Field usage determined by MODE bit.

Figure 7 details the possible sequencing states. Conditional sequencing defaults to an increment of the UMAR if the condition on the selected bus is not met. Note also that the conditional sequencing states can be merged to form a three-way branch. For instance,

$$\text{USDR}(64-70) = 0100011,$$

specifies a jump if $(Bi) = 0$, a skip if $(Bi) > 0$, and an increment otherwise.

USDR(71) causes the contents of the UMAR to be pushed into the micro-address stack. Any microinstruction may push the stack independent of its sequencing operation. This feature allows a convenient method of micro-subroutining without any additional microcode or execution time overhead.

USDR(72) selects the bus to be tested for conditional branching. Either bus 1 or bus 2 is always selected and unconditional branching is determined by the sequencing bits themselves.

USDR(73) determines whether the next microcycle will be finished or whether the processor will wait for an external continue from the virtual memory.

This microsequencing scheme makes microprogramming easier; however, it is somewhat limited. Experience has shown that the logical OR of certain of the bus conditions is very desirable. For example, one often wants to branch on a "greater-than or equal-to" condition. This can presently be accomplished by a branch on "less-than" but often at the cost of an extra microinstruction. Likewise, it is desirable to set and test (under microcontrol) a special status register which can remain static over microcycle boundaries. Present sequencing will be expanded to include some of these features.

As Husson [8] points out, the specification of the address of the next microinstruction and the available conditional branch tests have profound implications on the performance and the cost of the system.

3.6 Timing and Considerations

Microprocessor timing diagrams are shown in Figs. 8 and 9. As mentioned before, the CPU can execute either a short (fast) microcycle or a long (slow) microcycle. The fast cycle (430 ns) occurs when unconditional sequencing is required. A slow cycle (700 ns) is executed when the next microinstruction

Fig. 7. Microinstruction Sequence Control Bits

Fig. 8. Fast Microcycle Timing

Fig. 9. Slow Microcycle Timing

address is a function of the results of the execution of the current microinstruction. Tables II and III (which relate to the corresponding figures) describe the significance of particular instants of time during the microcycle. Table IV discusses the function of each timing pulse.

Note that each microcycle, either fast or slow, is divided into two parts. Pulses DPT1 through DPT5 are referenced to the leading edge of DPT1 and are initiated by it. Pulses DPT6 through DPT13 are referenced to the falling edge of the continue signal. By doing this, each microcycle can be broken by inhibiting the CONTINUE line. This is normally done when virtual memory references are executed, and it is this scheme which provides the asynchronous virtual memory operation. For example, if within a particular microinstruction a memory reference operation is performed, the microprogrammer has the option of either providing an internal continue (in which case the microcycle is not broken) or inhibiting the internal continue. If the latter is done, the microcycle is halted until the virtual memory or I/O device which has been addressed replies to the CPU with an external continue. Since all data registers are changed by pulses DPT6 through DPT13, no change in the data state of the CPU will have occurred.

Figure 10 is a block diagram of the main timing generator of the microprocessor. The continue selection logic performs the function just described. The continue pulse can either be generated internally by DPT3 or externally by a virtual memory continue.

In order that the current microinstruction control its own memory references, the microinstruction bits which control the continue selection logic (USDR(32-34) and USDR(73)) are stored in the LSDR (late storage data register); the LSDR is set at the end of a microcycle and holds these bits until the end of the next microcycle. Thus a memory reference operation, called for in a particular microcycle, doesn't actually occur until DPT3 of the next microcycle. This scheme was adopted so that the microprogrammer would be able to think in terms of performing a complete memory reference action within a single microcycle rather than have to break it up into two parts. If a late storage data register was not provided, microsequencing would effectively revert to the look-ahead scheme previously described.

Table II. FAST CYCLE TIMING INFORMATION

Time During Cycle	Significance
T0	Beginning of cycle.
T1	New microinstruction stable in USDR.
T2	Earliest time in cycle that an internal or external CONTINUE can occur. Also, time when logic knows whether cycle should be slow or fast (i.e., conditional or unconditional branch respectively).
T3	UMAR stable with address of next microinstruction. Also, time at which data is stable at outputs of lower busses.
T4	SPAR output stable if (SPAR) altered during microcycle. Also, time at which data is stable at outputs of upper busses.
T5	Data stable in destination registers.

Table III. SLOW CYCLE TIMING INFORMATION

Time During Cycle	Significance
T0	Beginning of cycle.
T1	New microinstruction stable in USDR.
T2	Earliest time in cycle that internal or external CONTINUE can occur. Also, time when logic knows whether cycle should be slow or fast (i.e., conditional or unconditional branch respectively).
T3	Time at which data is stable at outputs of lower busses.
T4	SPAR output stable if SPAR altered during microcycle. Also, time at which data is stable at outputs of upper busses.
T5	Data stable in destination registers. Also, conditional sequencing action is known by the logic.
T6	UMAR stable with address of next microinstruction.

Table IV. CPU TIMING PULSE FUNCTIONS

Timing Pulse	Function
DPT1	Memory request signal to control memory.
DPT2	Sets USDR with new microinstruction.
DPT3	Internal microcycle CONTINUE and/or virtual memory request.
DPT4*	Increments UMAR; pushes microaddress stack. Fast cycle only.
DPT5*	Sets UMAR; pops stack. Fast cycle only.
DPT6	Virtual memory control pulse.
DPT7	Increments SPAR; set manipulator registers.
DPT8*	Memory request signal (i.e., DPT1 of next microcycle).
DPT9	Sets SPAR.
DPT10	Sets destination registers and LSDR.
DPT11**	Increments UMAR; pushes microaddress stack. Slow cycle only.
DPT12**	Sets UMAR. Slow cycle only.
DPT13**	Memory request signal (i.e., DPT1 of next microcycle). Slow cycle only.

*Exists only during fast microcycles.

**Exists only during slow microcycles.

The beginning of each microcycle examines USDR(64-70) to determine whether a fast or slow cycle is to be executed. By time T2, this has been determined. If a fast cycle is to be performed, there is no need to wait for the results of the action of the microcycle to determine the next microinstruction, and the UMAR is either immediately incremented by DPT4 or set by DPT5. The fetch of the next microinstruction is then overlapped with the execution of the rest of the current microinstruction.

If a long cycle is performed, updating of the UMAR with the address of the next microinstruction must await the execution of the current cycle. In this case pulses DPT11, DPT12, and DPT13 are used, respectively, to increment and set the UMAR and fetch the next microword.

The execution times for the microcycles are due to several factors. First, each microinstruction involves a local-store read followed by one or two manipulations, followed by a local-store write plus microsequencing, memory reference, and register-to-register transfer operations. Second, the hardware is presently implemented with a mixture of standard TTL, high-speed TTL, and Schottky TTL logic. The standard speed TTL yielded some long propagation times--especially in the unary manipulator. Third, the main timing generator is a monostable ring made from standard TTL monostables. A standard pulsewidth of 50 ns was used because some of the logic would not respond to pulses shorter than 40 ns and also because the monostables used don't effectively generate shorter pulses. This pulse-width, when one considers the number of timing pulses per microcycle, accounts for a substantial amount of time. A higher speed monostable chain with home-built monostables could have been used. However, the reliability, stability, and ease of implementation gained by using this technique was felt to be highly desirable. Similarly, a high-speed clocked system could have been used for timing. Here, however, one would not have the ability to individually adjust each pulse with respect to both position within the microcycle and pulse-width. Also, asynchronous memory referencing would have been made more difficult. Finally, worst case propagation design was used in order to guarantee an operative system with a minimum of debugging. The adjustability of the timing generator will allow us to improve the time by "tweaking." We feel that by simply adjusting the timing, the cycle times can be reduced by from 8 to 12 percent.

By selectively using additional Schottky logic and redesigning the timing generator, a significant additional timing reduction of 21-35 percent can be attained. This would reduce the fast cycle to about 228 ns.

3.7 Machine Level Instructions

AMP's machine instruction format can be almost anything which fits within multiples of 16-bit words. The microprocessor provides a convenient means of putting instructions into a specific 16-bit format containing 3 fields (see Fig. 11). In this format an operation code can be placed in the "O" or "R" fields. These fields can also be used for an extension of an operation code. The "R" and "C" fields can be used for passing parameters between the machine language and the microsubroutines which execute the machine code. These fields are independently routed from the INR register to the UMAR and the busses. For instance, transferring the "O" field to the UMAR sets the UMAR to an address in the range 0 to 63. Similarly, putting the "C" field into bus 2 puts a positive two's complement number in the range 0 through 31 into that bus. As shown in Fig. 11, this word can be the header word of a variable word length instruction. This is possible because the instructions are executed in microcode. Words A1, A2, etc. may or may not exist for a specific instruction. What's more, these words need not be contiguous.

The UMAR may be set with (IROF) as a number between 0 and 63 or (IRRF) as a number between 64 and 95. This makes it convenient to organize the control memory as shown in Fig. 12. Transferring the contents of the "O" field to the UMAR amounts to a 64-way branch into the microbranch table. Similarly, transferring the contents of the "R" field into UMAR amounts to a 32-way branch into the extended branch table. The word in the microbranch table or extended branch table then becomes the first word of the microsubroutine which executes the particular machine instruction. The "O" field bit pattern is the microbranch table address. Each microinstruction in a branch table must then unconditionally jump to the next executable instruction of the microsubroutine which is located somewhere beyond location 95. (Note that the UMAR addresses 96-bit words (of which 74 are presently used for the microcontrol word) in the control memory over the range 0 to 2047.

Fig. 11. AMP Machine Instruction

Fig. 12. Control Memory Organization
for Using INR Fields

The control memory can also be addressed as part of virtual memory as 16-bit words in the range 48,128 to 64,511.) Using the hardware defined "O", "R", and "C" fields limits the generality of the microprocessor.

The three nearly equal fields per word represent a good compromise between field capacity and the number of fields per word. Multiregister variable word length machine languages are possible, and yield a wide variety of machine instruction possibilities. AMP field definition is similar to that of the QM1 "microinstruction" [3]. The QM1 "nanoinstruction" corresponds to AMP microinstruction.

The use of variable field definition controlled by either the microinstruction or by mask registers was considered. The hardware required to implement the selection and the skewing of fields was substantial, and the cost (in microcode bits) of direct microinstruction control was prohibitive.

3.8 Ease of Hardware Modifications

The processor was designed as a research tool; therefore, ease of hardware modifications and additions was considered to be of great importance. Hardware additions fall principally into two areas: additions to the CPU (e.g., additional general-purpose registers, manipulators, and local stores) and in the area of additional memory and peripherals.

The internal busses are very important when additions are made to the CPU. The busses are made up of one-of-eight data selectors. At present the internal busses have unused inputs. With the present number of bus microcontrol bits, additions can be made. The busses may be further expanded in groups of four, eight, or sixteen; however, this would require additional microcontrol word bits. Additional general-purpose registers, manipulators, or local stores will require input bus space and expansion of the microcontrol word. The control memory may be horizontally expanded to 96 bits by the addition of a plug-in board. Similarly, it may also be vertically expanded to 2048 words.

Any number of peripheral devices (any device which communicates with the CPU using the address bus, output bus, and input bus) up to the number of available peripheral addresses in virtual memory may be added. At present this is 27,648 words.

To facilitate hardware modifications and additions, an average integrated circuit packing density (on wirewrap boards) of 75 percent was used as a design target. The physical structure will allow additional hardware so that the present machine could be easily doubled in size.

3.9 Manipulators

Three ALU's exist within AMP, the unary, binary, and the byte manipulators. The unary manipulator is available to both microinstruction modes. Its function is to perform operations on single operands (e.g., increments, decrements, shifts, etc.). The binary manipulator is available during word-mode microinstructions. It performs arithmetic and logic operations on two operands. The byte manipulator is available to I/O-mode microinstructions. Its purpose is to facilitate byte operations within the word-organized processor. It performs byte-to-word and word-to-byte transformations (e.g., packing, unpacking, swapping, etc.). Because of the microinstruction mode concept employed in AMP, two manipulators are available during any given microinstruction. The CPU bussing structure cannot conveniently support more than two manipulators.

The philosophy of treating the manipulators on the basis of the number of operands is that the execution of code often deals with two classes of operations concurrently. For example, binary operations may be used for processing data while indexing type operations may be used for bookkeeping purposes. Manipulations on lists and arrays are typical examples of this process.

An alternate scheme which was considered involved having two identical manipulators each having unary and binary capabilities. This approach was not used because propagation time and cost were too great.

Schemes for the efficient use of multiple manipulators as suggested by Kleir and Ramamoorthy [9] or as described by Tomasulo [10] are not implemented in the hardware. This is, however, a fertile area for investigation.

The effectiveness of this particular arrangement of manipulators is discussed in subsection 5.2.4.

3.10 Operation of Peripherals

Peripherals are defined to be those devices which have microprocessor virtual memory addresses assigned to them. The peripherals in AMP are the 1.75 microsecond core, the 750 ns core, a vector generator, a disc controller, an electrostatic printer/plotter controller, the PDP-7 computer accumulator, and the control memory.

Communication between the microprocessor CPU and peripherals is by means of the address bus, the output bus, the input bus, and four control pulses. Typical peripheral communication paths are shown in Fig. 13. Four bits of USDR are stored during each microcycle in the late microstorage data register. Two of these bits ANDed with DPT3 and DPT6 produce the peripheral control pulses, IØREQ and IØST, respectively. A third bit of LSDR is converted to the control level, R/W. The fourth bit of USDR is used in the CPU timing as an internal continue bit.

Communication between the CPU and peripherals is asynchronous. The operation is shown in Fig. 14. After the contents of busses BA and BO have stabilized, the CPU issues a single timing pulse IØREQ after which the CPU timing chain is interrupted. Each peripheral monitors the address bus at the time of IØREQ. The addressed peripheral then performs the data transfer required. At the end of the data transfer, the peripheral issues a device continue pulse which restarts the CPU timing chain. The direction of transfer is determined by R/W.

To produce the above-described data transfer, the microprogrammer would place a peripheral address on the address bus and then issue the commands READ, or WRITE, and WAIT. For operating a device such as core memory, which requires only an address and one data transfer for its operation, the use of the I/O request pulse, the READ/WRITE line, and the device continue pulse is sufficient. However, a peripheral such as the disc controller requires several data transfers and a start pulse for its operation. The start pulse is provided by a bit in the late storage data register together with the CPU timing pulse DPT6. This is sufficient for communication between the CPU and all peripherals. However, considerable processing time may be wasted because the CPU is stopped between I/O request pulse time and the device continue time. This time amounts to several potential microcycle times. To take

Fig. 13. Typical Peripheral Communication Paths

Fig. 14. Peripheral Timing

advantage of peripheral processing time in the CPU, the microprogrammer can issue a read or write command and issue the corresponding wait command at any later point in his program. There are two restrictions on this type of operation. First, the information from the peripheral may not be available during the microinstructions processed between read or write and wait. Secondly, if more than one peripheral device is addressed before a wait command, the device that finishes its cycle first will cause the CPU timing chain to commence regardless of the order of the addressing of peripherals. Therefore, using the delayed wait feature requires an intimate knowledge of the relative response times of peripheral devices.

4. SOFTWARE

The present software support for AMP consists of a microassembler and an expandable macroassembler. Both assemblers were written in PL/I.

4.1 Microassembler

The microassembler is discussed in another document [11]; it is briefly mentioned here for completeness. The assembler has many of the capabilities of a conventional assembler, but it is also capable of multiple specifications per microword. The microlanguage is at a level close to that of the actual hardware. The following is an example of a single microinstruction with a descriptive explanation.

Example 1

SPE27<-A&B>SP010-B1->SPE0,MDRO	*NAND OF SPE27 & SP010-JSPE0 & MDRO
INCREMENT SPAR	*(SPAR)+1->SPAR
<ADD1>RA-B2->RA,BMAR	*(RA)+1->RA & BMAR
IF B1=0 GOTO #FRED	*CONDITIONAL JUMP
IF B1 ODD SKIP	*CONDITIONAL SKIP
PUSH	*CURRENT MICROADDRESS TO STACK

SPE27 and SP010 are locations in the even and odd local stores respectively. SPAR is the local store address register. SPE0 is a local store register and is addressed by SPAR. The two conditional sequencing operations effect a three-way branch.

All manipulator designations are enclosed by "<" and ">" symbols and the source busses are implied. Thus, <A&B> is a binary manipulator operation with SPE27 and SP010 routed to the manipulator over busses 3 and 4 respectively. The result is routed over bus 1 as specified. Similarly, <ADD1> is a unary manipulator operation with RA routed to the manipulator over bus 5.

4.2 Macroassembler

Initial machine and assembly languages have been defined for the microprocessor, and an expandable macroassembler (AMPMAC) has been written to support them. The languages can be modified as needed.

This software is the subject of a future document. It is mentioned here to build a frame of reference for the following section.

5. EXPERIENCES

5.1 Machine and Assembly Language

Presently, a limited instruction set exists for the microprocessor. As mentioned previously, the groundwork has been laid for modifying and expanding the languages as necessary.

Implementing the machine language at the microcode level involves the use of microsubroutines that are called by the machine instructions via the branch and extended branch tables located in the control memory (see Section 3.7). Microsubroutine linking is performed via a microsupervisor which fetches machine instructions and handles interrupts. The microsubroutines have been coded for execution speed rather than control memory efficiency.

Another approach that could have been used for microsubroutine linking was to eliminate the microsupervisor and allow each microsubroutine to fetch the next machine level instruction. This scheme was likely to execute more quickly than the previous one, but it would have been less efficient in terms of its use of control memory. However, because of AMP's internal parallelism, a possibility exists within this approach for improving control memory utilization. By efficiently locating the necessary micro-operations within the microsubroutine a machine level fetch may be distributed within the microsubroutine without the use of additional memory or execution time. The effect would be equivalent to overlapping the machine level fetch, decode, and execution times.

An interesting capability becomes available in designing the assembly language when one has access to three levels of coding. The question is continually raised as to which level (micro, machine, or assembly) to place certain features and burdens. For instance, the relationship between these three levels of code could be isomorphic. This arrangement would require a minimal assembler and simple microsubroutine calls with a unique microsubroutine for each assembly level instruction. Parameters may or may not be passed between the language levels.

Alternatively, a number of specific assembly level instructions could map to a single versatile machine instruction which calls a single microsubroutine. Parameter passing between assembly and machine code would be

minimal, but a large number of parameters would need to be passed between the machine instruction and the corresponding microsubroutine. In this case, the assembler is more complex, and the execution time of the code is longer because of the overhead associated with passing parameters. In terms of memory space, the machine level code will tend to be inefficient because a large number of parameters will have to accompany each instruction; however, the microcode will be minimized due to the small number of microsubroutines.

The following generalizations can be made. If parameter passing is required between the machine language and the microsubroutines, the implications are that execution time will be longer and memory space required for machine code will be large, but control memory requirements will be minimal. Conversely, if no parameter passing is done, execution time will be fast and the memory space required for the machine code will be small, but more micro-control memory will be needed.

Our present language contains versatile assembly- and machine-level instructions which pass a substantial number of parameters. This approach requires a minimum of microcode and provides us with a lot of versatility within a limited instruction set. The language consists of 40 assembly level instructions and it is implemented by approximately 300 microinstructions including the microsupervisor.

By dealing with all three language levels, the designer has the freedom to optimize for running time, memory efficiency, ease of assembler design, application programming, etc. As Cook and Flynn [5] point out, vocabularies with expansive instructions may be created.

5.2 Microcode Analysis

A graphics applications system has been developed. In this system a PDP-7 is used for data input and for handling human interactions, while AMP is used for data manipulations and the control of graphics peripherals.

In order to be able to evaluate microcode support of the assembly level language and also to investigate the direct support of the system at the microprogramming level, alternate programs were written in assembly language and directly in microcode for the AMP related systems functions. Thus the

AMP programming for the graphic system was implemented in two ways: (1) via an intermediate assembly language, and (2) directly as problem-oriented microcode.

The total microcode provides a sample size of approximately 350 microinstructions for statistical analysis. The two approaches allow some interesting comparisons to be made.

The analysis reported here is primarily static and is therefore subject to the limitations of this type of approach. Still, this analysis, reinforced in many cases by intuitive feelings, does provide some insights.

The concept of "instruction mix" at the microinstruction level has little or no meaning for horizontally structured microprogrammable machines because of the versatility of each microinstruction. Measures of parallelism and hardware use tell us something about the efficient use of the microprocessor, machine organization, future design, methods of microprogramming support, and etc. The rest of this section presents a series of histograms analyzing parallelism and resource utilization.

Before proceeding, one additional point should be made. The applications system under consideration is not necessarily typical. Indications are that AMP is "loafing" much of the time and is spending a significant portion of its active time performing I/O. A "number crunching" problem might yield different results.

5.2.1. Micro-operations per Microinstruction. Figure 15 is a micro-operation histogram (see definition of "micro-operation" in Section 2). The number of micro-operations in each microinstruction is a measure of the parallelism imbedded in the microcode. The mean value is equal to 2.62 micro-operations per microinstruction. The mean value for the problem-oriented microcode is equal to 2.72, while the mean value for the microcode supporting the machine language is equal to 2.58. While the histogram of Fig. 15 is for the sum of all of the code, the distributions for the two separate bodies of microcode did not differ significantly.

It could be argued that with a 74-bit microword, a considerably larger number of micro-operations per microcycle should be able to be executed. For instance, a single micro-operation as defined in this paper is as

Fig. 15. Microoperation Histogram

powerful as a machine instruction of a 16-bit minicomputer. Therefore, a 74-bit microinstruction should have a mean value of 4.6 micro-operations. It must be remembered, however, that the 16-bit machine instruction, like the vertical microinstruction mentioned previously, is highly encoded in order to maximize its efficiency. It was also pointed out that encoding severely limits instruction versatility. There is a direct trade-off between versatility and instruction efficiency.

5.2.2. Microinstruction Bit Utilization. Another interesting statistic involves bit utilization within the microinstruction. Figure 16 is a histogram providing this information. Bit utilization can also be thought of as a measure of parallelism, but it is also a measure of the overall CPU resource utilization. The mean value for all microinstructions is 29.7 bits per microword. The mean value for the problem-oriented microcode is 31.1 and the mean value for the microcode supporting the machine language is 29.2. As with micro-operations, the problem-oriented microcode fares slightly better. However, there is no significant difference in either the means or the distributions.

5.2.3. Microprocessor Resource Utilization. To see whether some of the factors limiting parallelism involved CPU resource utilization, we examined the use of the busses, manipulators, local store, registers, etc. Bus utilization is shown in Fig. 17. Note that the simultaneous use of both upper busses is reasonably high; it is almost the same as the simultaneous use of both upper busses when at least one of the lower busses is not used. In fact, the ratio of these last two numbers is approximately .98. This implies that some imbalance exists between the upper and lower busses. The imbalance could be improved by changing the organization to improve the utilization of bus 4.

Manipulator utilization is another item of interest. Figure 18 shows this. Over 30 percent of the microinstructions do not utilize the manipulator and it appears that the manipulators are not a limiting factor.

Local store utilization is shown in Fig. 19. In both cases the distribution is about the same, with the problem-oriented microcode faring slightly better. Although Fig. 19 doesn't show it, the microcode supporting

Fig. 16. Microword Bit Utilization

Fig. 17. Bus Utilization

Fig. 18. Manipulator Utilization

(a)

(b)

Fig. 19. Local Store Utilization

the machine language had a much higher utilization of local store addressing via the local storage address register. This is to be expected since the machine language is a multiregister type.

It is significant to note that the mean number of local store references per microinstruction is approximately one, even though 25 to 30 percent of the time no local store is addressed. This indicates that the method of local storage addressing is effective.

Investigation of microsequencing and especially conditional microsequencing shows that long cycles represent 16 percent of the microinstructions. Microinstructions which contain only sequencing operations and therefore in some sense are wasted, occur 3.5 percent of the time. Three-way branches are used less than one percent of the time as are microstack operations. The general feeling is that these figures are likely to improve with microprogrammer experience and with the addition of certain hardware modifications.

Figure 20 shows an interesting variation between problem-oriented microcode and the microcode supporting the machine language. Here the problem-oriented microcode needs a large number of constants but the microjumps are relatively few. For the machine language microcode the converse is true. This is not surprising since significant numbers of parameters are passed between the machine code and the microcode; thus few constants have to be derived for the microcode. The large number of jumps is due to the branch table and the large number of small microsubroutines in the microcode that supports the machine language. With the program-oriented microcode the subroutines are few and no branch table exists; however, constants are not passed to the microcode and must be generated within the subroutine.

5.2.4. An Execution Time Comparison. Figure 21 is a comparison of execution times for matrix addition problems. The subject matrices were 126×126 and had 16-bit integer elements. The execution times shown are for 1000 matrix additions. Care was taken to ensure that a linear relationship existed between the number of matrix operations and total execution time for each type of code.

Fig. 20. Jump Address/Immediate Field Utilization

Fig. 21. Matrix Addition Execution Times

The four cases shown are defined as follows:

- Case 1: Direct microcode approach. In this case, the process was completely micro-encoded.
- Case 2: Intermediate language (AMPMAC) with an expansive "multiple add and move" instruction.
- Case 3: Intermediate language (AMPMAC) without "multiple add and move." Machine level subroutines were coded to use indexing.
- Case 4: Intermediate language (AMPMAC) without "multiple add and move." Machine level subroutines were coded to use indirect addressing.

In order to interpret Fig. 21, some further definition is required. Let

T_t = total execution time,

T_c = actual time spent in computation,

T_i = microcode inefficiency due to nonexpansive machine level instruction, and

T_f = machine instruction fetch time.

Then

$$T_t = T_c + T_i + T_f \quad .$$

Furthermore, for the direct microcode approach (Case 1),

$$T_i = T_f = 0 \quad .$$

It can be seen that the machine instruction fetching overhead is substantial where the language is not very expansive (Case 3 and Case 4). In addition, code inefficiencies exist due to microlinkage overhead, and also because the short microsubroutines associated with simple machine instructions prevent the effective use of machine parallelism.

To provide a comparison, the same problem was executed by a fast assembly level program on an IBM System 360/75. CPU time on the Model 75 was approximately 4 percent greater than the execution time on AMP.

2

6. CONCLUSION

AMP is the result of initial research into low-level central processing unit parallelism and horizontal microcontrol. Its primary purpose is as a tool for further research in these and associated areas. As such, our design goal, to create a versatile and usable machine with easily modifiable and expandable hardware was accomplished.

A number of design features contribute to the versatility of the machine. A multiple bus structure with each bus having a multiple-input/multiple-output capability allows other resources within the CPU to be connected in a variety of ways. The low-level parallelism provided by the multiple ALU's, the register arrangement, and the local store addressing scheme adds to the versatility of the microinstruction. The use of virtual memory--especially for peripherals, is also a contributing factor.

A primary design feature which makes the machine easily usable at a microcode level is the "microoperation" concept. This feature required a more complex (and consequently slower) microcycle than other machines that have been either proposed or implemented [5], [3]. The virtual memory arrangement and its asynchronism makes memory referencing (and particularly I/O operations) very convenient for the microprogrammer. Another concession to ease of microprogramming was current word sequencing.

The machine was made hardware modifiable with the inclusion of expandable bus structures, expandable control word and memory size, the virtual memory scheme, and the construction techniques employed in building the machine.

Many of the design decisions were necessary compromises due to a finite research budget and limited manpower. With a larger microinstruction we could have eliminated the "MODE" feature and increased the number of busses and local stores. The versatility of the microsequencing could also have been improved by including a branch test register, a greater number of conditions on which to do microbranching, and providing such things as conditional stack operations. A faster microcycle would have been desirable. This would have required a faster control memory, a higher performance timing chain, and faster logic. All of these things would have impacted both

the budget and the manpower allocated to the project. The machine was designed, however, so that microcycle timing might be easily changed, and consequently the machine speed may be improved at a later date. Probabilistic sequencing would also have been included.

Initial CPU resource utilization analysis tends to show AMP to be a "balanced" machine. That is, hardware resource utilization tends to be relatively even. However, utilization figures are not as significant as the degree of low level parallelism which can be achieved. This is especially true today because of the low cost of hardware. It is no longer as economically effective as it once was to maximize CPU resource utilization.

The concept of micro-operations per microinstruction is a meaningful measure of low-level parallelism. The mean value of 2.62 can be increased by increasing microprocedure overlap. This, in turn, implies a greater level of interaction between microinstructions.

Improved organization and expanded hardware can relieve the limitations that are due to resource and location dependencies.* A number of hardware modifications are being made with this in mind. Microcode optimization strategies are being investigated to better utilize the resources that are available.

Experiences with AMP so far indicate that it is desirable to eliminate as many levels of intermediate language as possible and to design microcode to directly relate to more expansive machine instructions. On large blocks of microcode, the above dependencies may be more easily eliminated and the CPU parallelism more effectively utilized. The overhead associated with machine level instructions is reduced in an expansive instruction set.

* Location dependencies arise from placing information in registers or local-store locations from which the data is not directly available for further processing. These dependencies may occur due to a lack of forethought or because of conflicts between microprocedures.

APPENDIX A: MANIPULATOR FUNCTIONS

BYTE MANIPULATOR CONTROL BITS

CONTROL WORD BITS 38 39 40			OPERATION
0	0	0	<p>Halves of input word on Bus 4 swapped</p>
0	0	1	<p>First half of input word on Bus 4 unpacked</p>
0	1	0	<p>Second half of input word on Bus 4 unpacked</p>
0	1	1	<p>First halves of input words on Busses 3 & 4 packed</p>
1	0	0	<p>Second halves of input word on Busses 3 & 4 packed</p>
1	0	1	Input on Bus 3 passed through
1	1	0	Input on Bus 4 passed through
1	1	1	Output of zero provided

UNARY MANIPULATOR CONTROL BITS

CONTROL WORD BITS					OPERATION
41	42	43	44	45	
0	0	0	0	0	Clear Bit 15 of input on Bus 5
0	0	0	0	1	" " 14 " " " "
0	0	0	1	0	" " 13 " " " "
0	0	0	1	1	" " 12 " " " "
0	0	1	0	0	" " 11 " " " "
0	0	1	0	1	" " 10 " " " "
0	0	1	1	0	" " 9 " " " "
0	0	1	1	1	" " 8 " " " "
0	1	0	0	0	Set Bit 15 of input on Bus 5
0	1	0	0	1	" " 14 " " " "
0	1	0	1	0	" " 13 " " " "
0	1	0	1	1	" " 12 " " " "
0	1	1	0	0	" " 11 " " " "
0	1	1	0	1	" " 10 " " " "
0	1	1	1	0	" " 9 " " " "
0	1	1	1	1	" " 8 " " " "
1	0	0	0	0	Illegal Code
1	0	0	0	1	Subtract 1 from input on Bus 5
1	0	0	1	0	" 2 " " " "
1	0	0	1	1	Shift input on Bus 5 left 1 bit
1	0	1	0	0	" " " " " 2 bits
1	0	1	0	1	" " " " " 4 "
1	0	1	1	0	Rotate input on Bus 5 left 1 bit
1	0	1	1	1	Output of all one's provided
1	1	0	0	0	Input on Bus 5 passed through
1	1	0	0	1	Add 1 to input on Bus 5
1	1	0	1	0	" 2 " " " "
1	1	0	1	1	Shift input on Bus 5 right 1 bit
1	1	1	0	0	" " " " " 2 bits
1	1	1	0	1	" " " " " 4 "
1	1	1	1	0	Rotate input on Bus 5 right 1 bit
1	1	1	1	1	Output of all one's provided

BINARY MANIPULATOR CONTROL BITS

CONTROL WORD BIT PATTERN						T Y P E	OPERATION	
35	36	37	38	39	40		A $\equiv$ Bus 3	B $\equiv$ Bus 4
$\neg C$	M	S0	S1	S2	S3			
0	0	0	0	0	0	A	A+1	
0	0	0	0	0	1	A	A+(A&B)+1	
0	0	0	0	1	0	A	A+(A& $\neg$ B)+1	
0	0	0	0	1	1	A	A+A+1	
0	0	0	1	0	0	A	(A $\neg$ B)+1	
0	0	0	1	0	1	A	(A $\neg$ B)+(A&B)+1	
0	0	0	1	1	0	A	A-B	
0	0	0	1	1	1	A	(A $\neg$ B)+A+1	
0	0	1	0	0	0	A	(A B)+1	
0	0	1	0	0	1	A	A+B+1	
0	0	1	0	1	0	A	(A B)+(A& $\neg$ B)+1	
0	0	1	0	1	1	A	(A B)+A+1	
0	0	1	1	0	0	A	Zero output	
0	0	1	1	0	1	L	A&B	
0	0	1	1	1	0	L	A& $\neg$ B	
0	0	1	1	1	1	L	A	
0*	1	0	0	0	0	L	$\neg$ A	
0*	1	0	0	0	1	L	$\neg$ A B	
0*	1	0	0	1	0	L	$\neg$ (A&B)	
0*	1	0	0	1	1	L	All 1's output	
0*	1	0	1	0	0	L	$\neg$ A&B	
0*	1	0	1	0	1	L	B	

BINARY MANIPULATOR CONTROL BITS (cont'd)

CONTROL WORD BIT PATTERN						T Y P E	OPERATION	
35	36	37	38	39	40		A≡Bus 3	B≡Bus 4
$\neg C$	M	S0	S1	S2	S3			
0*	1	0	1	1	0	L	$A \mid B$	
0*	1	0	1	1	1	L	$A \mid B$	
0*	1	1	0	0	0	L	$\neg(A \mid B)$	
0*	1	1	0	0	1	L	$\neg(A \mid B)$	
0*	1	1	0	1	0	L	$\neg B$	
0*	1	1	0	1	1	L	$A \mid \neg B$	
0*	1	1	1	0	0	L	Same as 001100	
0*	1	1	1	0	1	L	Same as 001101	
0*	1	1	1	1	0	L	Same as 001110	
0*	1	1	1	1	1	L	Same as 001111	
1	0	0	0	0	0	A	Same as 001111	
1	0	0	0	0	1	A	$A+(A\&B)$	
1	0	0	0	1	0	A	$A+(A\&\neg B)$	
1	0	0	0	1	1	A	$A+A$	
1	0	0	1	0	0	A	Same as 011011	
1	0	0	1	0	1	A	$(A \mid \neg B)+(A\&B)$	
1	0	0	1	1	0	A	$A-B-1$	
1	0	0	1	1	1	A	$(A \mid \neg B)+A$	
1	0	1	0	0	0	A	Same as 010111	
1	0	1	0	0	1	A	$A+B$	
1	0	1	0	1	0	A	$(A \mid B)+(A\&\neg B)$	
1	0	1	0	1	1	A	$(A \mid B)+A$	
1	0	1	1	0	0	A	minus 1 (2's complement)	
1	0	1	1	0	1	A	$(A\&B)-1$	
1	0	1	1	1	0	A	$(A\&\neg B)-1$	

BINARY MANIPULATOR CONTROL BITS (cont'd)

CONTROL WORD BIT PATTERN						T Y P E	OPERATION	
35	36	37	38	39	40		A $\equiv$ Bus 3	B $\equiv$ Bus 4
$\neg C$	M	S0	S1	S2	S3			
1	0	1	1	1	1	A	A-1	
1*	1	0	0	0	0	L	Same as 010000	
1*	1	0	0	0	1	L	" " 010001	
1*	1	0	0	1	0	L	" " 010010	
1*	1	0	0	1	1	L	" " 010011	
1*	1	0	1	0	0	L	" " 010100	
1*	1	0	1	0	1	L	" " 010101	
1*	1	0	1	1	0	L	" " 010110	
1*	1	0	1	1	1	L	" " 010111	
1*	1	1	0	0	0	L	" " 011000	
1*	1	1	0	0	1	L	" " 011001	
1*	1	1	0	1	0	L	" " 011010	
1*	1	1	0	1	1	L	" " 011011	
1*	1	1	1	0	0	L	" " 011100 and 001100	
1*	1	1	1	0	1	L	" " 011101 and 001101	
1*	1	1	1	1	0	L	" " 011110 and 001110	
1*	1	1	1	1	1	L	" " 011111 and 001111	

* Bit 35 ($\neg C$) ignored by AMP

Type A: arithmetic operation

Type L: logic operation

| $\equiv$ logical OR

|| $\equiv$ logical exclusive OR

APPENDIX B: MICROINSTRUCTION SUBFIELD DEFINITIONS

USE	BITS	BREAKDOWN
MODE	0	0 : BYTE and I/O MODE 1 : WORD MODE
IMF	2-11	IMF or Micro-Address Field
RA Input	12-13	00 : NULL 01 : Select 10 : Select B1 11 : Select "OR" of B1 & B2
BMAR Input	14-15	00 : NULL 01 : Select B4 10 : Select B2 11 : Select "OR" of B2 & B4
B1	16-18	000: Select Byte Box 001: Select Unary Box 010: Select Unary Box Extended 011: Zeros 100: Select Binary Box 101: Select MDRI 110: Zeros 111: Zeros
B2	19-21	000: Select Byte Box 001: Select Unary Box 010: Select Binary Box 011: Select IMF 100: Select IRCF 101: Select IA 110: Zeros 111: Zeros
B3	22-24	000: Select RA 001: Select RB 010: Select SPO 011: Select SPE or SPEO 100: Select IMF 101: Select IRRF 110: Zeros 111: Zeros
B4	25-27	000: Select RA 001: Select RB 010: Select SPO 011: Select SPE or SPEO 100: Select IMR 101: Select ITR 110: Zeros 111: Zeros

Microinstruction Subfield Definitions (cont'd)

USE	BITS	BREAKDOWN	
B5	28-30	000: Select RA 001: Select RB 010: Select SPO 011: Select SPE or SPEO 100: Select SPAR 101: Zeros 110: Select RSW 111: Zeros	
MDRO Input	31	0 : NULL 1 : Select B1	
RB Input	32-33	00 : NULL 01 : Select B2 10 : Select B1 11 : Select "OR" of B1 & B2	Word Mode
INR	34	0 : NULL 1 : Select B1	
Binary Box	35-40	See Binary Manipulator Control Bits Chart	
I/O Request	32	0 : NULL 1 : I/O Request	
START	33	0 : NULL 1 : START	
READ/ WRITE	34	0 : NULL 1 : WRITE	Byte & I/O Mode
IMR Input	35	0 : NULL 1 : Select B1	
UMAR Input	36-37	00 : NULL 01 : Select IRRF 10 : Select IRØF 11 :	
Byte Box	38-40	See Byte Manipulator Control Bits Chart	
Unary Box	41-45	See Unary Manipulator Control Bits Chart	
SPAR Control	46-47	00 : NULL 01 : B3 to SPAR 10 : SPAR ← SPAR+1 11 :	

Microinstruction Subfield Definitions (cont'd)

USE	BITS	BREAKDOWN
SPO Address or IMF	48-52	Field contains: Extended IMF or ($0 \leq \text{xxxxx}_2 \leq 31$) SPO Address
SPO Input	53-54	00 : NULL 01 : Select B2 10 : Select B1 11 : Select "OR" of B1 & B2
SPE Address	55-59	SPE Address ($0 \leq \text{xxxxx}_2 \leq 31$)
SPE Input	60-61	00 : NULL 01 : Select B2 10 : Select B1 11 : Select "OR" of B1 & B2
Scratch Pad Source Address	62	0 : Select SPO and SPE 1 : Select SPAR
JUMP	64-66	000: $\text{UMAR} \leftarrow (\text{UMAR})+1$ 001: JUMP IF < 0 010: JUMP IF $= 0$ 011: JUMP IF > 0 100: JUMP IF ALL ONES 101: JUMP IF ODD 110: UNUSED 111: JUMP UNCONDITIONALLY
RETURN (POP)	67	0 : NULL 1 : POP [$\text{UMAR} \leftarrow (\text{Return Address}) + 1$]
SKIP	68-70	Same as JUMP field
ST.RTN. ADD. (PUSH)	71	0 : NULL 1 : PUSH (Store Return Address)
Test Bus	72	0 : Test B1 1 : Test B2
Continue	73	0 : WAIT 1 : CONTINUE
Unused	74-79	Unused

ACKNOWLEDGMENTS

The authors would like to thank Robert Clark, who wrote the micro-assembler; William Snow, for the wirewrap and debugging program; Edward Jones, for his overall programming assistance; and David Jacobsohn, for many helpful suggestions. Their contributions have been significant to the overall effort.

REFERENCES

- [1] W. P. Lidinsky, "MIRAGE, A Microprogrammable Interactive Raster Graphics Equipment," Proc. IEEE Comput. Soc. Conf., pp. 15-16, Sept. 1971.
- [2] M. V. Wilkes, "The Growth of Interest in Microprogramming: A Literature Survey," *Computing Surveys*, vol. 1, pp. 139-145, Sept. 1969.
- [3] R. F. Rosin, G. Frieder, and R. H. Eckhouse, Jr., "An Environment for Research in Microprogramming and Emulation," Preprints 4th Annual Workshop on Microprogramming, Sept. 1971.
- [4] M. J. Flynn and R. F. Rosin, "Microprogramming: An Introduction and a Viewpoint," *IEEE Trans. Comput.*, vol. C-20, pp. 727-731, July 1971.
- [5] R. W. Cook and M. J. Flynn, "System Design of a Dynamic Microprocessor," *IEEE Trans. Comput.*, vol. C-19, pp. 213-222, Mar. 1970.
- [6] S. R. Redfield, "A Study in Microprogrammed Processors: A Medium Sized Microprogrammed Processor," *IEEE Trans. Comput.*, vol. C-20, pp. 743-750, July 1971.
- [7] H. W. Lawson, Jr. and B. K. Smith, "Functional Characteristics of a Multilingual Processor," *IEEE Trans. Comput.*, vol. C-20, pp. 732-742, July 1971.
- [8] S. S. Husson, *Microprogramming: Principles and Practices*. Englewood Cliffs, N.J.: Prentice-Hall, 1970.
- [9] R. J. Kleir and C. V. Ramamoorthy, "Optimization Strategies for Microprograms," *IEEE Trans. Comput.*, vol. C-20, pp. 783-794, July 1971.
- [10] R. M. Romasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal* 11, pp. 25-33, 1967.
- [11] R. K. Clark, "Mirager, the 'Best-Yet' Approach for Horizontal Microprogramming," Proc. 1972 Assn. Comput. Mach. Conf., 1972.