PDP-11/40E Microprogramming Reference Manual *

S. H. Fuller, G. T. Almes, W. H. Broadley, C. L. Forgy,
P. L. Karlton, V. R. Lesser, J. R. Teter

16-Jan-76

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

---

* This work was supported by the Advanced Research Projects Agency of the Department of Defense under contract no. F44620-73-C-0074 and monitored by the Air Force Office of Scientific Research.

TABLE OF CONTENTS

1. Introduction 1
   - 1.1 Design Objectives and Constraints 1
   - 1.2 Synopsis 2
   - 1.3 Acknowledgements 4

2. Hardware Description 5
   - 2.1 The Basic PDP-11/40 5
   - 2.2 The Extensions to the PDP-11/40 17

3. MICRO/40 Assembler 38
   - 3.1 Introduction 38
   - 3.2 Field Assignments 40
   - 3.3 Pseudo Operators 41
   - 3.4 The Assignment Statement 43
   - 3.5 Examples 45
   - 3.6 Using MICRO 48

4. Microprocessor Simulator 49
   - 4.1 Summary 49
   - 4.2 Commands 49
   - 4.3 Inconsistencies 53

5. Programming Techniques 54
   - 5.1 Timing 54
   - 5.2 B Constants 55
   - 5.3 Unibus Control 55
   - 5.4 Flow of Control 57
   - 5.5 RD Bus Details 60
   - 5.6 The Last Word 61

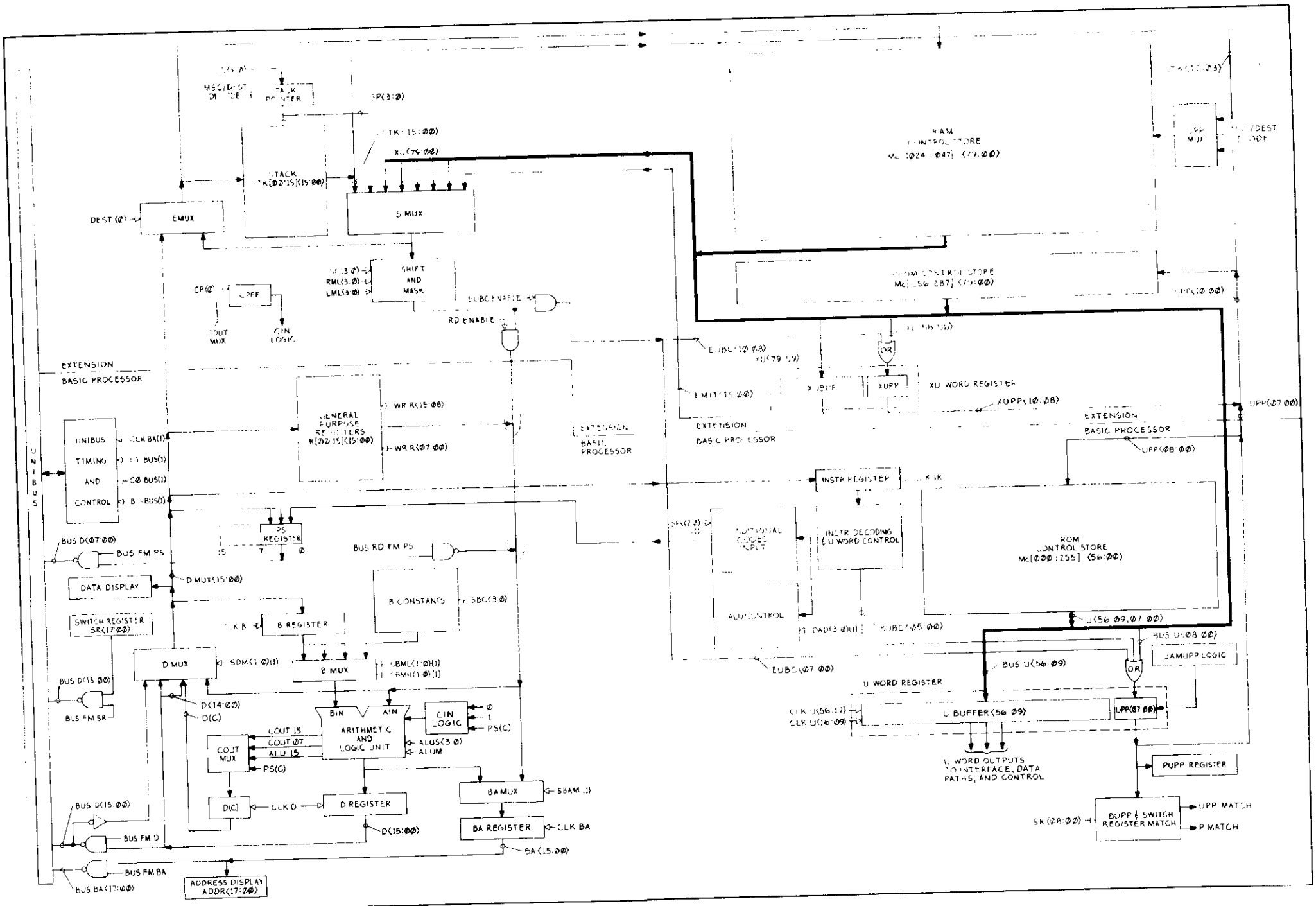
Appendices

A. Microinstruction Format 62
   - A.1 The Word 62
   - A.2 Address Space 75
   - A.3 DEST/MSC Functions 77

B. Standard ROM in MICRO/40 78

C. The Bootstrap PROM 88

D. Vector Instruction Set 90


UNIBUS
UNIBUS TIMING AND CONTROL
CLK BA(1)
C1 BUS(1)
C0 BUS(1)
B BUS(1)
EXTENSION BASIC PROCESSOR
GENERAL PURPOSE REGISTERS R(00:15)(15:00)
WR R(15:08)
WR R(07:00)
EXTENSION BASIC PROCESSOR
EMUX
DEST (2)
STACK POINTER
SP(3:0)
STACK STK(00:15)(15:00)
STK(15:00)
X(79:00)
S MUX
SHIFT AND MASK
SF(3:0)
RML(3:0)
LML(3:0)
CP(2)
CPEN
COUT MUX
CIN LOGIC
EUBC ENABLE
RD ENABLE
RAM CONTROL STORE MC[024:047] (79:00)
ROM CONTROL STORE MC[256:287] (79:00)
UPP MUX
UPP(10:00)
EUBC(10:08)
XU(79:09)
OR
EMIT(15:00)
XUBUF
XUPP
XU WORD REGISTER
XUPP(10:08)
EXTENSION BASIC PROCESSOR
UPP(07:00)
EXTENSION BASIC PROCESSOR
UPP(08:00)
INSTR REGISTER
CLK IR
SEN(2:0)
ADDITIONAL CODES INPUT
INSTR DECODING & U WORD CONTROL
ROM CONTROL STORE MC[000:255] (56:00)
U(56:09,07:00)
BUS U(08:00)
JAMUPP LOGIC
BUS U(56:09)
OR
EUBC(07:00)
U WORD REGISTER
CLK U(56:17)
CLK U(16:09)
U BUFFER(56:09)
UPP(07:00)
U WORD OUTPUTS TO INTERFACE, DATA PATHS, AND CONTROL
PUPP REGISTER
BUPP & SWITCH REGISTER MATCH
SR(08:00)
UPP MATCH
P MATCH
PS REGISTER
BUS RD FM PS
BUS D(07:00)
BUS FM PS
DATA DISPLAY
D MUX(15:00)
SWITCH REGISTER SR(17:00)
B CONSTANTS
SBC(3:0)
CLK B
B REGISTER
D MUX
SDM(1:0)(1)
B MUX
SBML(1:0)(1)
SBMH(1:0)(1)
BUS D(15:00)
BUS FM SR
D(14:00)
D(C)
BIN
AIN
ARITHMETIC AND LOGIC UNIT
CIN LOGIC
0
1
PS(C)
ALUS(3:0)
ALUM
COUT MUX
COUT 15
COUT 07
ALU 15
PS(C)
D(C)
CLK D
D REGISTER
D(15:00)
BUS D(15:00)
BUS FM D
BUS FM BA
BUS BA(17:00)
ADDRESS DISPLAY ADDR(17:00)
BA MUX
SBAM(1)
BA REGISTER
CLK BA
BA(15:00)

1. Introduction

The PDP-11/40E was developed at Carnegie-Mellon University because no commercially available machine could meet pressing needs for a processor that could both support user microprogramming for specialized applications and effectively emulate the standard PDP-11 instruction set and I/O structure. The PDP-11/40 was available, however; it was implemented on a microprogrammable processor and, in addition to its read-only control store, had room in the processor for an extended control store. The PDP-11/40E is a standard PDP-11/40 processor that has been extended to include the following features:

1. 1024 words of writable microprogram control storage. This is in addition to the 256 words of read-only storage in the basic PDP-11/40 processor. Each word is 80 bits wide in the writable control store.

2. A general purpose mask-shift unit. It can be used either to assist the PDP-11/40's basic ALU in the manipulation of data or to extract arbitrary fields from words for branch control in the microprogram.

3. A subroutine facility at the microprogram level.

4. A 16-bit literal field to assist in the generation of masks and constants.

5. A 16 word stack for temporary variables and microsubroutine return addresses.

1.1 Design Objectives and Constraints

There were several major design considerations that influenced the design of the writable control store option for the PDP-11/40:

A) Compatibility Requirement with the Basic PDP-11/40 Processor

1. Fit all additional hardware into the basic PDP-11/40 hardware cabinet with minimum of modification to the basic processor, its power supply, and cooling; limit added hardware to approximately 120 integrated circuit packages.

2. Ability to run the standard PDP-11 emulator in the writable control store.

3. Retain the basic cycle times of the standard processor.

B) Writable Control Store Requirements

1. Must easily be read and written under microprogram control.

2. Should be addressable as 16-bit memory, usable as a scratchpad memory.

3. The size of the control store should be large enough to hold the entire emulator of a sophisticated machine architecture.

C) General Emulation Requirements

1. General field extraction rather than built in extraction of fields peculiar to the PDP-11 instruction set.

2. General branch table jumping based on arithmetic rather than built in combinatorial logic peculiar to the PDP-11 instruction set.

3. Subroutine linkage mechanism for convenient coding of large microprograms.

4. Arbitrary constants for arithmetic and addressing.

5. Convenient implementation of multiple precision arithmetic.

These features have been added to the PDP-11/40 with an absolute minimum modification to the PDP-11/40 and are included in the processor by preempting the space that had been reserved in the processor for the extended instruction set (EIS) and floating point instruction set (FIS) options. Both EIS and FIS options have been reprogrammed for the writable control store and hence are still available on the PDP-11/40E by loading their microcode into the control store. The only other modifications to a standard PDP-11/40 are minor and are detailed in the PDP-11/40E Engineering Documentation [CMU75]. The writable control store has no effect on the other options available on the PDP-11/40; i.e. memory management, stack limit register, maintenance console, and the line frequency interrupt clock.

1.2 Synopsis

This reference manual begins in the next section with a complete description of the PDP-11/40E microprocessor. All the registers, multiplexors, data paths and control lines available to the microprogrammer are described. For details of the microprocessor concerning implementation, installation, and maintenance questions see [CMU75]. Section 3 describes MICRO/40, a cross-assembler for the PDP-11/40E microprocessor, which runs on a DECsystem10. It includes a macro facility, compound statements, and assignment statements. Although these features have been added to assist the programmer, an important principle of MICRO/40 is that it allows a programmer to construct any microinstruction (however baroque). This is an essential feature in 'horizontal' microprocessors such as the PDP-11/40E if we are to write efficient emulators for varying machine architectures. Section 4 discusses the simulator of the PDP-11/40E microprocessor which is also available on the DECsystem10. The simulator's primary purpose is to provide a reasonable debugging

environment for microprogram development. Section 5 discusses the microprogramming techniques used at CMU and describes some of the idiosyncracies that characterize the PDP-11/40E. The final section includes a number of examples to illustrate the various features of the PDP-11/40E and MICRO/40. Appendix A gives a definitive description of the microinstruction fields and their use. Following this comes Appendix B: the MICRO/40 version of the standard PDP-11 emulator residing in the 250 words of read-only memory in the standard microprocessor. The MICRO/40 version of the PDP-11 emulator is a good starting point for alterations, as opposed to extensions, of the PDP-11 instruction set. version of the PDP-11 emulator as a starting point. Appendix C lists the bootstrap PROM used at CMU during the summer of 1975.

The following DEC documentation may be of assistance in the use of the PDP-11/40E:

KD11-A Processor Maintainence Manual. DEC-11-HKDAA-A-D (1972). This manual describes the standard PDP-11/40 (i.e. the KD11-A) at the same level of detail that the PDP-11/40E is described in this manual. It also includes a detailed discussion of the 256 word PDP-11 emulation program. The KD11 Processor Engineering Drawings are frequently referenced by the KD11-A Processor Maintainence Manual and should be available when using this manual.

PDP-11/40 Processor Handbook. (1972). This is the general reference manual for the PDP-11/40 system as seen by the PDP-11 programmer. Although it has no information on the PDP-11/40 microprocessor, it is the definitive document on the PDP-11 architecture as it has been implemented on the PDP-11/40. The handbook also contains an introduction to the Unibus and describes the functions of all the standard internal processor options of the PDP-11/40.

PDP-11 Peripherals Handbook. (1975). This is the best source of information on the theory and operation of the Unibus.

KE11-E and KE11-F Instruction Set Options Manual. DEC-11-HKEFA-A-D (1973). Describes the standard EIS and FIS option. Chapter 4 provides a concise summary of the basic KD11-A processor and may be of use to the PDP-11/40E microprogrammer for this reason.

PDP-11/40, 11/35 System Manual. DEC-11-H40SA-B-D (1974). A general introduction to the standard PDP-11/40 system, including the installation, operation, and programming of the PDP-11. It is of limited use to the microprogrammer.

In addition to these DEC publications, we refer interested readers to a recent book by Andrew Tanenbaum, Structured Computer Organization, Prentice-Hall, 1976, which discusses the standard PDP-11/40 microprocessor.

1.3 Acknowledgements

The PDP-11/40E project began in July, 1973, has spanned several years, and has benefitted from many individuals. The original PDP-11/40 was donated to CMU by Digital Equipment Corporation and G. Bell, J. Levy, and C. Kaman (all from Digital) provided helpful technical review of the project. W. Broadley, S. Fuller, V. Lesser, B. Rosen, and J. Teter designed the writable control store extension for the PDP-11/40 during the fall of 1973. The first PDP-11/40E became operational in October 1974. L. Forgy wrote the simulator described in section 4. P. Karlton wrote the MICRO/40 assembler described in section 3. G. Almes, P. Drongowski, N. Jain, and R. Modi, as the first serious users of the PDP-11/40E, endured many hardware difficulties during the first year of its operation and were very helpful in locating and correcting several design flaws.

2. Hardware Description

2.1 The Basic PDP-11/40

The basic PDP-11/40 * consists of a horizontal microprocessor with a microinstruction 56 bits wide **. Almost 256 instructions are necessary to implement the standard PDP-11 instruction set. Some of the internal register and data paths are general in nature and would be used in any 16 bit processor. But some areas, particularly instruction decoding, were implemented specifically for the PDP-11 instruction set and are not of a general purpose nature.

The register-transfer block diagram of Figure 2.1 (see the end of section 2) is discussed below in three major functional groupings: interface, data paths, and microprogram control. All of the components in each of these segments are covered in detail in sections 2.1.1 through 2.1.3. In addition, Table 2.1 contains a listing of all components on the block diagram and includes a brief physical description as well as related inputs and outputs. This abbreviated summary can be used as a quick reference once the more detailed description of the block diagram is understood, or it can be used for a quick overview of the KD11-A processor by those who are already familiar with PDP-11 processors and microprogramming techniques.

2.1.1 Interface Logic

The first section of the processor shown in the block diagram is the interface logic which is used to interconnect the KD11-A processor with other components of the PDP-11/40 System such as the programmer's console, Unibus, etc. Each of the functional blocks shown on the interface portion of the block diagram is covered in the following subsections.

---

* Much of section 2.1 of this reference manual was adapted from the KD11-A Processor Maintainence Manual[1972], a publication of Digital Equipment Corporation. The material in this section is the sole responsibility of Carnegie-Mellon University. For more complete information on the basic PDP-11/40 microprocessor, refer directly to the KD11-A Processor Maintenence Manual.

** Throughout this manual, the term microprocessor denotes any microprogrammable processor. We do not imply the use of LSI microcomputer technology as in the LSI-11 processor

2.1.1.1 KY11-D Programmer's Console

The KY11-D Programmer's Console is an integral part of the PDP-11/40 system and provides the programmer with a direct system interface. The console allows the user to start, stop, load, modify, or continue a program. Console displays indicate data and address flow for monitoring processor operations. The console logic that is considered to be a part of the processor interface section includes the switch register, the data display, the address display, and the console control. All functions of the KY11-D console are implemented with microcode routines.

The switch register is located on the KY11-D console and consists of the manually-operated switches gated through drivers to the Unibus. The microprogram addresses the switch register during console operation and, decoding the address, enables the driver gates, which place the value set in the switch register onto the Unibus.

The data display indicates the output of the processor data multiplexer which gates information from a variety of sources within the processor, and also gates data from the Unibus. The display consists of indicator lights, mounted in the programmer's console, that are connected to the processor by cables. The output line of the data multiplexer (D MUX <15:0>) always controls the display. However, since the multiplexer can select multiple inputs onto the output line, information can be displayed from a variety of sources.

The address display indicates the contents of the processor bus address register (BA register). Note that there is no multiplexing involved with the address display which was the case with the data display. Although it is possible to load specific data into the bus address register for different situations arising in the logic flow, the contents of the bus address register is always shown by the address display.

The console control logic is associated with the programmer's console operational switches that provide such manual functions as START, HALT, LOAD ADDRESS, EXAMINE, DEPOSIT, and CONTINUE. Primary console control is handled by the processor by means of both the microprogram and combinational logic flag flip-flops. The microprogram senses switch activation and branches to the specific routine required, depending on which switch has been used. The flags accommodate the special needs of the START and CONTINUE switch sequences as well as the incrementation requirements of consecutive EXAMINE or DEPOSIT sequences.

2.1.1.2 Unibus Timing and Control

The Unibus timing and control logic provides the required processor regulation of the Unibus, controls data transfer functions, bus ownership functions, and other miscellaneous functions. The control logic includes drivers and receivers for Unibus signal lines as well as timing and priority logic. Combinational logic, pulse circuits, and discrete flip-flops provide control for data transfers (DATI, DATIP, DATO, DATOB) between the processor and the bus with associated error checking (odd address, stack overflow) and correction (data time-out). The logic also provides the gates and signals needed for the processor to respond once it has been addressed from the bus.

In addition to the data transfer function, the Unibus timing and control logic provides the necessary control for bus ownership, transfer of bus ownership for non-processor requests (NPRs) and bus requests (BRs), and the time-out function for non-response conditions. The logic also provides power fail timing related to BUS AC LO, BUS DC LO, and BUS INIT signals. Combinational logic, which includes a number of one-shot timing circuits, sequences these signals for power on and power off conditions.

The microprogram control interfaces directly with the Unibus timing and control logic. The start or error checking flip-flops are loaded, either directly or conditionally, from the microinstruction. Acceptance of bus data and the deactivation of MSYN occur as a function of the next microinstruction after a DATI or DATIP transfer operation. The processor transmits address and data information to the bus under control of the microprogram. Note, however, that bus ownership, as well as the power fail logic, operates asynchronously and is independent from the microprogram.

The interface portion of the processor contains both bus transmitters and bus receivers so that processor and Unibus signals are compatible. The outputs of the bus address (BA) register, and the D register, the processor status (PS) register, and the switch register all have individual sets of transmitters (drivers) to place their contents on the Unibus. Inputs to the processor from the bus are gated through the bus receiver to the D multiplexer which then routes the signals to the proper component within the data paths.

2.1.2 Data Paths

The data paths portion of the KD11-A Processor manipulates, stores, and routes data within the processor. The prime element of the data path logic is the arithmetic logic unit (ALU) which operates, both logically and arithmetically, upon input data from the interface portion of the processor. The ALU and all other components in the processor data paths are described in the following paragraphs.

There are twenty-one major registers in the KD11-A processor. Sixteen of these registers are implemented as an addressable 'scratch pad' memory, used for temporary storage and implementing macro-program visible registers. The five remaining registers are the B register, (an ALU input), the D register (the ALU output), the BA register (the bus address), the PS register (the macro-processor status), and the IR register (the macro-program instruction register).

The scratch pad registers are the primary source of operands for the arithmetic and logic unit (ALU). These operands either come directly from an instruction source or destination mode operand, or they are stored in a scratch pad register during address calculations. In either case, the ALU receives a direct input from the BUS RD <15:0> line. This input is referred to as the "A input." The characteristics of the scratch pad register affect the data path structures in that only one address may be accessed at a time and simultaneous read and write operations are not permissible. In order to provide the two ALU operands (when both operands come from the scratch pad register), it is necessary to provide temporary storage. This storage is provided

by the B register. The contents of the B register can be fed through the B multiplexer into the B input of the ALU.

2.1.2.1 Data Paths, Multiplexers, and Registers

Since the A input to the ALU is from BUS RD<15:0>, it can supply operands from the scratchpad registers, the PS word, or the processor extension. Further, since any combination of these registers can be gated onto the BUS RD during execution of any microinstruction, it is possible to OR the contents of several registers onto the BUS RD lines, as shown by the dotted OR gates in Figure 2.1.

The B input comes from the B multiplexer (B MUX) which receives its input from either the B constants or the B register. The B register, in turn, receives its input from the D multiplexer which has four possible inputs. Therefore, the B input to the ALU comes from a variety of sources with two levels of multiplexing. These various inputs are discussed in the following paragraphs.

The four inputs to the D multiplexer are: Unibus data lines BUS D <15:0> (which permits the processor to receive operands from other devices in the system), the BUS RD <15:0> lines, the output of the D register (which permits the result of a previous arithmetic operation to be used as an operand), and the right shifted output of the D register.

The D multiplexer output can be stored in the B register which in turn can be fed to the ALU by means of the B multiplexer. It should be noted that the BUS RD signal can be fed through the D multiplexer into the B register. This data path is of special interest in the machine instruction for the register-to-register operations where the B input of the ALU must come from the scratch pad register. For example, if both desired operands are stored in the scratch pad register, the first operand passes through the D multiplexer into the B register for storage. Then the second operand can be fed to the A input of the ALU while the first operand is fed to the B input by means of the B multiplexer.

The B constants, which are applied through the B multiplexer to the ALU, provide elementary values (such as 1 and 2 for incrementing or decrementing scratchpad registers). They also provide other values such as the switch register address, more complex constants such as trap vectors or masks for manipulating instruction offsets, and the conditional constants which are a function of machine status and jumper selection. Thus, they are not truly 'constant'.

The B input to the ALU can be either the B constants value or one of four possible functions of the B register:

B register: the contents of the register are applied directly to the ALU.

B extend: the B register contents are gated so that B<7> (MSB of the low-order byte) provides sign extension for the high-order byte. Note that in this case the value in the high-order byte is either all 1s or

all 0s depending upon B<7> of the B register. The low-order byte of the B register is applied directly to the ALU.

Byte duplication: either the low-order byte or the high-order byte may be duplicated. Therefore, the B input of the ALU equals either B <15:8> concatenated with B <15:8> or B <7:0> concatenated with B <7:0>.

Byte swapping: the high-order and low-order bytes are exchanged. Hence, the B input of the ALU equals B <7:0> concatenated with B <15:8>.

2.1.2.2 Arithmetic Logic Unit

The arithmetic logic unit (ALU) is the heart of the data path logic. It performs 16 Boolean operations and 16 arithmetic operations on two 16-bit words. The ALU is controlled by six input signals. One signal (ALUM H) selects either the logic or arithmetic mode of operation. Four signals (ALUS0 through ALUS3) select the desired function. The sixth signal is the output of the carry (CIN) logic. Basically, the ALU receives two 16-bit words as inputs (AIN and BIN) and performs the operation selected by the six control signals. The output of the ALU is used for Unibus addresses and data, and internal processor registers such as the scratch pad register or the processor status register. The output of the ALU is stored in the D register and/or the BA register for use in subsequent microinstructions.

Operation of the ALU is also determined by the carry-in (CIN logic) and carry-out (COUT MUX) signals. The carry-in signal does not come directly from the microprogrammed word, but is a function of the microprogrammed word and the conditions (usually the instruction register) that are enabled at specific locations in the microprogram flow.

The carry-out multiplexer (COUT MUX) provides multiplexing of the specific carry information normally used in the PDP-11. The signals that can be selected are: COUT 15, COUT 07, ALU 15, and PS(C). The COUT 15 signal represents the carry from a word operation and the COUT 07 signal represents the carry from a byte operation. These signals are used for condition code inputs and rotate/shift operations. The ALU 15 signal is the bit 15 output of the ALU which is used for rotate/shift operations. The PS(C) signal is the carry bit from the processor status register. The signal selected by the COUT MUX is clocked into an extension of the D register which is called D(C). This storage extension is used in rotate/shift operations and in certain arithmetic operations.

2.1.2.3 Decoding

The address and data decoding logic is a combinational logic network that decodes the output of both the D and BA registers. When the D register output is decoded, the decoder senses whether or not the output (for both byte and word segments) is zero (D <15:0> = 0). The BA (bus address) register is decoded to determine if a 'processor address' has occurred, or if the address is less than specified values. It should be noted that the decoding logic decodes the BA register and not the Unibus address. A 'processor address' is the Unibus address of the internal registers of the processor. If the decoded address is the address of the PS register (Unibus address 177776) or the console switch register (177570), then either PS ADRS H or SR ADRS H is true. If the decoded address is less than the specified value when using the macro stack pointer, then a stack overflow violation may occur and BOVFL signal is true. Stack limit errors are either yellow zone (warning) or red zone (fatal) indications.

2.1.2.4 PS Register

The processor status (PS) register is an 8-bit register that stores information of the macro processor; it includes the current priority of the processor (bits <7:5>), the result of the previous operation (condition codes bits N,Z,V,C), and an indicator for detecting the execution of an instruction to be trapped during program debugging (T bit). The status register is located between the two primary data paths: D MUX <15:0> and BUS RD <15:0>. The register is loaded from the D MUX. The condition codes control logic provides other inputs to the N,Z,V, and C bits. The register output is either gated directly onto the Unibus (in cases where the processor has addressed the PS register as an absolute Unibus address) or is gated onto the BUS RD <15:0> line for use by the processor data paths. This latter case is used, for example, by the condition code instructions which alter the contents of the processor status register.

2.1.2.5 Register REG

The 16 internal processor registers are referred to as the "scratch pad register". Eight of these are macro programmable general registers which include the program counter (PC) and stack pointer (SP). In the KD11-A processor, the additional eight registers (not accessible to the program) are used for a variety of functions. Such functions include: intermediate address (TEMP), source and destination data (SOURCE, DEST), a copy of the instruction register, (IR), the last interrupt vector address (VECT), registers for console operation (TEMPC,ADRSC), and a stack pointer for operation of the KT11-D Memory Management Option (SP USER).

In summation, the data path logic is the fundamental section of the processor and performs data storage, modification, and routing functions. The other two sections of the processor (interface and control) exist primarily to support the data path logic.

2.1.3 Control Logic

The final section of Figure 2.1 is the microprogram control logic which provides the required control signals for the data path logic and the interface logic. The primary element of the control logic is the read-only memory (ROM) which provides the various microinstructions. The bits in each microinstruction (U WORD), in turn, control the basic machine operation. Other elements within the control section include address and address modification logic that receives inputs from the ROM, the instruction register with associated decoding logic, various processor flags, and basic machine timing and flag control logic. The format of the microinstruction is shown in Figure 2.2. See Appendix A for a more complete description of the microinstruction format.

When a macro instruction is fetched from an external data storage location, the instruction enters the processor from the Unibus, passes through the D MUX, and is loaded into the instruction register under control of the microprogram. The output of the instruction register is decoded by combinational logic (IR DECODE) to provide microbranching (basic microbranch code, BUBC) for several branch conditions and the discrete auxiliary signals required by condition code and ALU control logic. The logic associated with the instruction register and condition codes will be discussed first because of its interaction with the data paths section. We will then proceed to the discussion of the basic microcontrol unit.

2.1.3.1 Condition Codes Input

The condition codes are used to store information about the results of each instruction so that this information can be used by subsequent instructions. The information recorded in the condition code bits (N,Z,V,C) of the processor status register differ for each instruction type and often differ for the part of the instruction being executed. Furthermore, the information to be recorded can vary for different classes of instructions. The condition codes logic is combinational logic that alters the condition codes during the latter part of an instruction cycle. During this time, condition codes are combinations of data register contents, overflow situations, etc. The decoded output of the IR DECODE logic and the select processor status (SPS) code of the microinstruction determine which conditions are to be presented as the data input to the processor status register. In addition, the SPS code determines when the processor status register should be loaded directly from the D MUX.

2.1.3.2 ALU Control

The ALU control combinational logic receives the DAD (discrete alteration of data) code from the microinstruction as a function of the IR decode logic and combines it with the explicit ALU control through the SALU field of the microinstruction. In general, the DAD code directly alters operation of the ALU; however, during the latter part of an instruction, where common instruction flow paths exist for several instructions, the DAD code is combined with the instruction register to alter operation of the ALU.

2.1.3.3 Flag Control

The flag control logic is closely related to the IR decode logic because certain instructions require specific flags such as WAIT and HALT. Flip-flops within the flag control logic interact directly with the microbranch logic to provide the required branch conditions in the microprogram flow to provide flag service.

2.1.3.4 U Branch Control

The next control store address (next machine state) is dependent on a number of previous conditions. The purpose of the microinstruction branch control (U BRANCH CONTROL) logic and the branch microtest (BUT) multiplexer is to select the next proper machine state. The microbranch control provides some of the inputs to the branch microtest (BUT) decoding logic by combining the diverse instruction decoding of the instruction register and encoding it into two, three, four, or five bits of a microaddress alteration, called Basic MicroBranch Codes for specific BUTs (BUBC (BUT XX)). For most of the complicated branches, such as the first instruction branch or some of the subsequent source or destination instruction branches, these codes are fairly extensive. On the other hand, they may be fairly simple, consisting of only three bits or, in some cases, three bits of another BUT encoded with one special condition. This is particularly true with the INSTR 2 BUBC and the (BYTE and INSTR 2) BUBC.

2.1.3.5 BUT MUX

The branch microtest multiplexer (BUT MUX) selects sets of address alterations to alter data into the microprogram pointer (UPP) which points to the next ROM address. The BUT MUX provides a 5-bit output with the number of possible inputs on the lowest order bits being greater than the number of inputs that can be selected for the higher order bits. This corresponds to the fact that few of the branches involve all five or six bits of address alteration. There are a number of address alterations that involve only one bit, usually the lowest order bit.

The microbranch control logic provides wide branch encoding situations for instruction situations (INSTR1, INSTR 2, and INSTR 3). A 5-bit input is possible for the BUBC signal. In other cases, the instruction register itself may be used for a single BUBC bit code when a single bit chooses one of two different microaddresses. The flag control logic also provided certain inputs which alter only one bit of the microaddress.

The actual selection of which input (wide or narrow branch, branch on instruction, branch on flag) is to be used is determined by the microprogram branch field (UBF) of the microinstruction. The UBF field directly selects which inputs of the multiplexers are applied to the microaddress alteration logic (the OR gate on the block diagram). The UBF field is five bits wide and the 37_8 branch conditions are specified in Appendix A.

2.1.3.6 U Word Control ROM and U Word Reg

The heart of the control logic is the microinstruction control ROM which stores 256 56-bit words, each of which represents a different machine state of the processor. The ROM provides a wired OR output to enable easy expansion of the processor as required by the writable control store option.

The microinstruction output of the ROM is latched in a buffer register (U WORD REG). This permits one microinstruction to be used for machine control and selection of the next address while the ROM is obtaining the next microinstruction. Although this results in a faster processor, this implementation increases the complexity of the hardware and makes the processor somewhat more difficult to understand.

Each microinstruction from the ROM consists of a control portion and a next address portion. At the beginning of the current machine state, a ROM output microinstruction is clocked into the U WORD register. The bits in the control portion of the microinstruction select addresses and multiplexers and enable clocking gates (these gates enable clock pulses toward the end of the machine state). The bits in the address portion of the microinstruction access the ROM to obtain the next ROM word. At this point, the address is fixed in the microinstruction register and alteration for a BUT has not occurred.

The delay in using the buffer (U WORD register) is fixed by the settling time of the flip-flops (approximately 15-20 ns). This is significantly better than the 60-90 ns required for addressing the ROM. For this reason, the buffer takes the delayed output of the ROM, clocks it at the beginning of the machine state, and makes it almost immediately available (in that machine state) to the rest of the processor (data path, interface, and the microprogram control itself).

The clock for the U WORD register is taken directly from the basic processor clocking and is related to the clock length selection bits in the microinstruction control. The clock is a function of a machine cycle and is the last pulse edge of the previous machine cycle.

Each microinstruction is divided into two segments: address and control. The address portion of the word is represented by BUS U <8:0> which is the address of the next ROM word. The control portion is represented by BUS U <56:9> which includes the control bits for the microinstruction. The control bits are applied directly to the U WORD while the address bits pass through an OR gate to the microprogram pointer (UPP) portion of the U WORD.

The outputs of the U WORD register are diverse and are used throughout the processor. Outputs control the basic processor clock, microcontrol branching, and elements of the interface and data path. These outputs are indicated by the labels on the U WORD REG outputs.

2.1.3.7 Microaddress Alteration

Each microinstruction contains the address of the next microinstruction to be used by the processor. This address is referred to as the MicroProgram Field (UPF) of the ROM. If this address were always used unchanged, little attention would have to be given to it here. However, alterations to this address are made for branching purposes. Therefore, there must be a method of modifying and storing this address so that the next specified word can be fetched while the current microinstruction executes. As shown on the block diagram, the hardware used to perform these functions consists of the OR gates on the UPF output (BUS U <8:0>), the output of the BUT MUX, and the UPP register. The base address of the UPF can be altered by the BUT MUX inputs resulting in a different next ROM word address in the UPP register.

In discussing the addresses in the microaddress loop, it is important to realize that an altered next address has been stored in the UPP register and that alterations for the subsequent next address are fed to the OR gate. Both of these addresses are clocked simultaneously; therefore, the address fed through the OR gate is clocked into the UPP and the address that had been stored in the UPP is clocked out. Consequently, in any given microinstruction, the control portion of the U WORD is performing manipulations while the UPP address portion of that word is addressing the next ROM word. The last UPP contents address of the above present U WORD are stored in the Past MicroProgram Pointer (PUPP) for diagnostic purposes.

Another address in the address loop is the output of the ROM which has been selected by the next address in the UPP register. This address does not appear immediately in the machine cycle (as is the case for the UPP next address) because ROM access time is greater than flip-flop settling time. However, it is present about midway through the U WORD state. This ROM output address, which appears on BUS <8:0>, is a subsequent next address and is applied through the OR gate to the UPP register. The next microinstruction is becoming available across the entire ROM and is to be clocked in after the current machine state ends. If the subsequent next address is fixed (i.e., no branches are required), then there is no real difference between the address and control portion of the ROM/U WORD interface. If a microbranch is to occur, it must occur at this point before the subsequent next address is clocked into the fixed UPP register. The branch requires a subsequent next address with some 0s in it. It also requires the BUT MUX logic to input alterations to this address. Both of these occurrences require that the current microinstruction has enabled appropriate control bits in the address and control sections. Note that the microbranch test in a current word cannot alter the next word. However, it can alter the following word (the subsequent next word).

Consider, for example, the portion of microcode for the MOV instruction beginning at microinstruction MOV00 (see page 4 of the Flow Diagrams in the KD11 Processor Engineering Drawings or Appendix D, page 69, of this document). MOV00 is executed on any MOV or MOVB instruction with destination mode one, i.e. of the form MOV(B) src, @Rn. Since the destination mode is one, the BA register is set to R[DF]. Further action, however, depends on (1) whether the instruction is MOV or MOVB and, independently, (2) whether the source mode is zero or not. To determine this, a special microbranch, the BUT 22, is invoked. The instruction register is examined: if

the source mode is zero, a 1 is ORed into the BUT MUX; independently, if the instruction is MOVB, a 2 is ORed into the BUT MUX. As MOV00 and its BUT 22 are being executed, microinstruction MOV07 is being fetched unconditionally. MOV07 is then clocked into the U WORD register and executed. In the MOV00 sequence, it is essentially a noop; it is necessary, however, so the needed four-way branch can occur. The UPF field of MOV07 contains the address of MOV16, which is divisible by four (the address is 200_8, as it happens); as MOV07 is clocked into the U WORD register, the UPF field is ORed with the BUT MUX and clocked into the UPP register. Once MOV07 completes, the desired four-way branch is taken to either MOV16, MOV17, MOV14, or MOV13 (locations 200_8, 201_8, 202_8, or 203_8). The reader is encouraged to trace several examples in the standard ROM microcode; as he gains familiarity with it, the trickiness of the technique will diminish, but its subtle charm will persist.

2.1.3.8 JAMUPP Logic

The microprogramming address loop is also affected by the Jam MicroProgram Pointer (JAMUPP) logic which alters the sequential nature of the address loop. The JAMUPP logic provides a means of jamming an address into the microprogram pointer to modify the microprogram for certain conditions such as bus errors, stack overflow, auto restart, etc. This logic provides the next microinstruction address directly as a function of previous start or error conditions in the machine. The output of the JAMUPP logic directly sets or clears the UPP register flip-flops to establish the required address. This method differs from the normal NOT/OR inputs which are clocked into the UPP register flip-flops.

2.1.3.9 PUPP Register

The output of the UPP register is also fed to the PUPP (Past MicroProgram Pointer) register at each system clock. The PUPP register maintains a history of the previous microprogram pointer and displays its contents on the maintenance console. Note that the previous pointer indicates the current microinstruction address.

2.1.3.10 BUPP & SR Match

The output of the UPP register is also fed to the BUPP & SR MATCH logic which is used for maintenance purposes. This logic compares the contents of the UPP register (UPP <8:0>) with the low-order bits of the switch register (SR <8:0>) and generates a match signal when UPP <8:0> equals SR <8:0>. This match signal can be used as a sync signal to trigger an oscilloscope or can be used to stop the clock (halt the machine). For example, to obtain a strobe signal upon entering ROM address 234, this address would first be set in the switch register on the programmer's console. When the contents of the UPP register matched the switch register value, the clocking pulse ending that machine state would be enabled as a strobe signal. Because the UPP register contains the next ROM address, the pulse would occur at the end of the machine state just prior to the state of the address in the switch register.

2.1.3.11 Clock Control

The clock logic and related timing signals are basic to any processor. The clock signals that are generated are either used directly or are gated with enabling signals. These enabling signals are derived directly from either the microinstruction or from machine states (flags, flip-flops, Unibus states, etc.). Data transfers and processor initializations within the processor itself are synchronous; they occur at specific times within machine states. Three different clock cycles are provided by the logic: 140 ns, 200 ns, and 300 ns. This synchronous operation is designed for continuous running of the processor as the ROM sequences one microinstruction after another. The processor should, however, be considered as a combination of both synchronous operation and asynchronous operation. The asynchronous nature of the processor is due to the fact that, upon certain conditions, the clock is turned off and waits for a restart. An obvious turn-off situation is during Unibus data or bus ownership operations which are specified as asynchronous functions.

There are three functional elements that comprise the processor clock logic: the clock pulse generator, the clock control, and the clock enable gates.

2.1.3.12 Clock Pulse Generator

The clock pulse generator produces the system clock pulses when triggered by the clock control logic. These clock pulses are used throughout the processor and are combined with the enable signals of the ROM to act upon the three major segments of the processor (interface, data path, and microprogram control). There are three cycle lengthes generated by the clock pulse logic: CL1 (140 ns) cycle which generates a P1 pulse; CL2 (200 ns) cycle which generates a P2 pulse; and CL3 (300 ns) cycle which essentially combines the CL1 and CL2 cycles and consists of P2 and P3 pulses. The primary purpose of the CL3 cycle is to allow a scratch pad register to be used as the A-input in an ALU funtion and then loaded from the result of this ALU function. The specific cycle length (CL1, CL2, or CL3) for a microinstruction is determined by the clock control bits (CLK field).

2.1.3.13 Clock Control

The clock control logic consists of a clock (CLK) and an idle (IDLE) flip-flop. The CLK flip-flop provides a pulse, while the IDLE flip-flop drives the RUN console light and indicates when the processor is actively processing microinstructions.

2.1.3.14 Clock Enable Gates

The clock enable gates receive the pulses generated by the clock pulse generator. During each machine state, microcontrol bits control the passage of these clock pulses to specific registers. When it is desired to clock a register, the microcontrol word has the appropriate bit enabled and the clock pulse passes through the enable gate to the clock input of the specified register.

The flag control logic recognizes a variety of asynchronous conditions and changes the sequence of processor operations in response to these conditions. The logic consists of discrete flip-flops and combinational logic that determines sequencing of trap elements, trap instructions, and error traps. When any of these conditions occur, the processor enters a trap service sequence of microprogram states and the logic generates a trap vector that is used to transfer system control to a specific trap service program.

2.1.4 Interface of Extension to the Basic PDP-11/40

A basic constraint in the design of the PDP-11/40 extension was the set of processor signals that is readily available. All the inputs to the microinstruction register are available as well as the inputs and outputs of the microinstruction address register so that the extension can supply its own 56 bit microinstruction to control the basic data paths and registers. Additional registers and data paths are controlled by additional bits on the microinstruction. In fact, the extension supplies another 24 bits to control its functions to make a microinstruction of 80 bits. Besides the microinstruction lines, there are two data busses: the DMUX bus which supplies data from the processor and the Bus RD which supplies data to the processor. In a normal PDP-11/40, these lines provide an interface to the EIS/FIS options. The writable control store pre-empts these physical and logical positions. In addition, some twelve additional wires must be added to the backplane.

2.2 The Extensions to the PDP-11/40

As shown in Figure 2.1, central to the PDP-11/40E is the writable control store. It consists of 80 1024-bit bipolar random access bipolar memory integrated circuits (Fairchild's 93415). The memory has a dual mode organization: when supplying microinstructions it is a 1024 x 80 bit memory; when data is being written into or read from it it is organized as a 5120 x 16 bit memory. In the 1024 x 80 bit mode, the writable random access memory (RAM) feeds both the 56 bit microinstruction buffer (U<56:0>) in the basic processor and the 24 bit extended microinstruction buffer contained in the extension. The address multiplexor (UPP MUX) selects the microinstruction address from the processor (EUPP<8:0>) as well as three additional address bits in the XUPF register (added to expand the control address space from 512 words to 2048 words). In the 5120 x 16 bit mode, the UPP MUX selects the stack output for the address of the 16 bit word to be written or read. In a write operation, the data comes from the DMUX bus; in a read operation, the data goes to the BUS RD to the D register. Although upon examination of the flow chart, other paths and destinations seem possible, the tight timing considerations make the D register the only source and destination of RAM data.

Since the RAM consists of active elements, the writable control store loses its contents whenever power is removed. In order to have some basic code to be able to read, write and execute the code in the microinstruction RAM, a 32 word non-volatile programmable-read-only-memory (PROM) is included in the extension.

The stack, a 16 word x 16 bit memory, is either used as a push-pop stack for data or, in the case of reading or writing the RAM, as an address. The push/pop operation is under direct control of the microprogrammer. The data input is selected by the EMUX to be either the DMUX bus or the output of the SMUX. The stack output can be placed via the SMUX, shifter, and mask unit either on the BUS RD, or on the EUBC lines to modify the microinstruction address of the next microcycle. Directing the SMUX to select the EMIT field (the high order 16 bits of the XU Buffer) and writing the output of the SMUX onto the stack and then placing the output of the stack on the EUBC lines implements microsubroutine return addresses.

The shifter, operating on the output of the SMUX, implements a 0 to 15 position right rotate. The right/left mask unit allows the programmer to specify how many bits from the left and how many bits from the right will be masked off (made zero). The combination of the shifter and right/left mask allows the extraction of any contiguous n-bit field located anywhere in the result. The result can be placed either on the BUS RD as data or on the EUBC bus for multi-way branching.

2.2.1 The Extended Micro Instruction

The logical bit position assignments have been changed from the original DEC numbering scheme to improve the ease of programming. In particular, the next microinstruction address field (UPF) has been moved from U<7:0> to XU<55:48>. This field concatenated with the extended address field makes a new 11 bit address field

XUPF<10:0> = XU<58:48>

The first 256 words of the control store address space is the original ROM of the PDP-11/40. The next 32 words are the extension PROM. The next 736 words are not implemented and the upper 1024 words are the RAM control store.

The DEST/MSC field supplies a 5 bit operand which is decoded into a number of operations described in Appendix A. It can specify that the high order 16 bits of the microinstruction, the EMIT field, be treated as a 16 bit data word. When this happens, the logical functions of the high order 16 bits are disabled. The data path is from the XU Buffer through the SMUX and then either through the EMUX to the stack or through the mask/shifter to the BUS RD or the EUBC lines. The path to be taken depends on the particular DEST/MSC code.

When the EMIT field is not being used, the low order 12 bits (XU<75:64>) are divided into three 4-bit fields: one to specify the left mask limit, one to specify the right mask limit, and one to specify the shift count.

XU<76> is the carry propagate control. When it is on, it applies the output of the COUT MUX from the last time it was enabled to the carry input of the ALU, and, at the end of the instruction, stores the new value of the COUT MUX. This function is particularly useful for multiple precision arithmetic.

XU<77> is the stack push/pop enable. When the value is a one and a write into stack operation occurs, the pointer is first decremented, then the word is written. In a read operation, the pointer is incremented at the end of the instruction.

The carry out multiplexer (XU<79:78>) allows direct control of the ALU carry out multiplexor.

2.2.2 RAM READ-WRITE Operations

The timing on a READ/WRITE operation on the RAM is such that the microinstruction following the instruction with the RAM operation must be located in the extension PROM unit. This is primarily because the PROM address lines do not go through the UPPMUX. It also means that the PROM cannot be read as data. The address for read/write RAM operations is on the top of the stack. The address contained on the stack is a byte address to be consistent with other PDP-11 addresses even though RAM can only be accessed as 16-bit words. The 1024 word x 80 bit memory is divided up to provide a 5120 x 16 bit memory that has contiguous addresses. This is accomplished by taking STACK<14,2:1> and using the results as a group select. Thus the first 4096 words contain the low order 64 bits of each microinstruction and the next 1024 words contain the high order 16 bits from each microinstruction. STACK<12:3> specify which of the 1024 microinstructions is being addressed.

As soon as the microinstruction is decoded for a RAM operation, and the PROM is deselected, the UPPMUX selects the stack as the address source. On a write operation, the data is written after a delay of 100 nanoseconds from the start of the instruction. On a read, the data is presented to the BUS RD after the specified shifting and masking occur, and the D register is clocked at P2 time. At P2 time, the PROM is selected and the UPPMUX switches back to the UPP lines. Since P2 time marks the beginning of the read cycle for the next instruction, all RAM operation instructions must be CL3 cycles.

2.2.3 Entry into the Writable Control Store

Whenever the microprocessor decodes an unused operation code, it transfers control to location 0 of the extension PROM. (The PROM is actually enabled whenever the microinstruction address falls in the range 256:512.) At that time the instruction that caused the transfer is located in general register 13, the B register, and in the instruction register.

2.2.4 Unibus Control

The data for UNIBUS write operations is contained in the D register and the address is contained in the BA register. Data from READ operations goes through the DMUX. One of the consequences of this is that UNIBUS data cannot be read directly into the D register. Normal operation is to set C1 BUS, C0 BUS and BG BUS for the appropriate operations and set CLKOFF. The system will automatically delay bus operations until the bus is free. The clock is turned on at the end of the Unibus cycle or when CLKOFF is executed, whichever occurs last. MSYN is turned off at the next P1 or P3 cycle so the instruction following a CLKOFF operation should be either a P1 or P3 cycle.

Turning the clock off can also be used to await the end of the peripheral cycle on the UNIBUS (C1 BUS, C0 BUS, BG BUS = 010). The clocking of non-processor requests (NPR's) and interrupts are influenced by the microprogram. In particular, NPR's are clocked either by MSYN, BGBUS, SETCLK when the clock is off waiting for the end of a peripheral cycle, or clocking the instruction register. Break Requests, i.e. bus requests at priority levels 7, 6, 5, or 4, are strobed on BUT26, MSYN, and CLK IR. If you're in a long sequence of microinstructions without UNIBUS traffic, care must be taken to allow NPR's to occur. Break Requests are normally not as important and would require interrupt handing microcode in the writable control store or a return to the basic 11/40 microcode.

Table 2.1
PDP-11/40E Functional Components

Component	Description	Input	Output
ADDRESS Display	Indicator lights located on the KY11-D Programmer's Console.	Contents of the Bus Address (BA) Register.	Displays contents of the BA Register on console ADDRESS display.
Arithmetic Logic Unit (ALU)	Four 74181 IC chips and one 74182 IC chip provide a 16-bit arithmetic logic unit with a lookahead carry. Dependent on mode selected, can perform up to 16 logic functions and up to 16 arithmetic functions. (See ALU TABLE, print D-BD-KD11-A-BD.)	Data: AIN–16-bit wide input from buffered BUS RD bus BIN–16-bit wide input from B MUX CIN–carry insert to LSB of ALU from CIN logic Control: SALUM, SALU (03:00) 5-bit wide control that specifies ALU function.	Data: Provides 16-bit output to either the D Register or to the BA Register through the BA MUX. Status: COUT 7, COUT 15, ALU 15 to input of COUT multiplexer.
Arithmetic Logic Unit Control (ALU CONTROL)	One 8233 IC (dual 2-line to 1-line multiplexer) and combinational logic. Generates control signals that are used to specify the ALU function.	ALU control signals from: microword bits, IR decode logic, and external control (KE11-E).	Five control signals, SALUM, SALU (03:00) that select the ALU function to be performed.

Table 2.1 (Cont)
PDP-11/40E Functional Components

Component	Description	Input	Output
B Constants	Combinational logic network providing elemental values for incrementation and decrementation. Also provides more complex constants such as trap vectors and masks.	Constants generated are a function of the following inputs: SBC (03:00) from the microword. STPM (04:02) from the trap sensing logic.	Selected constants applied to the B MUX.
B Multiplexer (B MUX)	Eight 74153 multiplexer IC chips. Provides the means of selecting the data input to the B input (BIN) of the ALU.	Control of the high and low bytes are independent signals from the microword. Any one of the following inputs can be selected: a. BC (15:00) (B constants) b. B (15:00) (direct) c. B (15:08, 15:08) (duplicate upper byte of B Register) d. B (07:00, 07:00) (duplicate lower byte of B Register) e. B (07:00, 15:08) (swap bytes of B Register) f. B (15:08=7, 07:00) (sign extend lower byte of B Register)	Provides 16-bit wide input to the B input of the ALU.

Table 2.1 (Cont)
PDP-11/40E Functional Components

Component	Description	Input	Output
B Register	Four 74174 IC chips provide a 16-bit temporary storage register for the B input of the ALU.	Input is loaded from the output of the D MUX and is therefore dependent on the D MUX selection.	Provides a data input to the B MUX. This input (which is the B Register output) is partitioned into a high (15:08) and low (07:00) byte.
Bus Address Multiplexer (BA MUX)	Four 8233 multiplexer IC chips. The BA MUX loads the BA Register.	Receives 16-bit wide input from either the Register Data bus (BUS RD) or the output of the ALU. A single microword control signal selects one of the two possible inputs. A high signal selects the ALU.	A 16-bit wide output that is loaded into the Bus Address (BA) Register.
Bus Address Register (BA Register)	Four 74174 IC chips that form a 16-bit temporary storage register.	Receives a 16-bit wide input from the BA MUX.	Transmits a 16-bit address to the Unibus. This address is applied through bus drivers to bus address lines BUS BA (17:00). The address is also applied to the address display and is decoded for processor address response.

Table 2.1 (Cont)
PDP-11/40E Functional Components

Component	Description	Input	Output
Bus Register Data (BUS RD)	Four 74H04 IC chips that provide 16 inverters to establish proper input polarity for the A Input (AIN) of the ALU.	Receives input from three sources by means of a wired-OR bus: a. Scratch Pad Register data (16 bits) b. Processor Status Register (8 bits) c. External options (16 bits)	Output provides 16-bit data to either the A input (AIN) of the ALU or to the bus address (BA) multiplexer.
Branch Microtest Decode (BUT DECODE)	Network of combinational logic circuits that decodes the Microbranch Field (UBF) in each microword and generates auxiliary control signals.	UBF (04:00) from the microword.	Control signals, especially to the flag control logic.
Branch Microtest Multiplexer (BUT MUX)	Six multiplexer IC chips: a. three 16-line to 1-line type 74150 multiplexers b. two 8-line to 1-line type 74151 multiplexers c. one dual 4-line to 1-line type 74153 multiplexer	Any one of the following are selected by microword UBF (04:00) field: a. IR Register bits b. branch microbranch control signals c. IR decode signals d. machine status and flags	Control signals that allow modification of the microprogram field, UPF (07:00), prior to clocking the address into the UPP of the Microword Buffer (U WORD).

Table 2.1 (Cont)
PDP-11/40E Functional Components

Component	Description	Input	Output
Buffered Microprogram Pointer and Switch Register MATCH (BUPP & SR MATCH)	Nine exclusive-OR gates connected as an equivalence detector. Compares the contents of the Microprogram Pointer Register (UPP) with the Switch Register (SR) to generate a MATCH signal. The MATCH signal can be used to stop the clock during maintenance operation or to generate a scope synchronizing signal. Comparing the two registers permits stopping operation or monitoring operation at a specific ROM word.	BUPP (08:00) and SR (08:00)	UPP MATCH signals
Clock Control	Network of combinational logic circuits and delay line controls the CLK and IDLE flip-flops.	CLKOFF from the microword as well as various restart and continue signals.	Control signals to the clock pulse generator.
Clock Pulse Generator	Three delay lines selected by combinational logic circuits to generate the clock pulses specified by the current microword.	Pulse signal from clock control and the clock length signals CLKL0 and CLKL1 from the microword.	Timing pulses P1, P2, and P3. The RECLK signal which provides for continuous microword operation.

Table 2.1 (Cont)
PDP-11/40E Functional Components

Component	Description	Input	Output
Clock Enable Gates	Combinational logic network that routes clock outputs to the INTERFACE, DATA PATHS, and MICRO-CONTROL portions of the processor.	Timing pulse P1, P2, or P3 from the clock pulse generator. Various clock enable signals: CLKIR, CLKBA, CLKB, CLKD, WRII, WRL bits from the current microword.	Various clock signals. (CLK IR, CLK D, CLK BA, etc.).
D Multiplexer (D MUX)	Eight 74153 multiplexer IC chips.	A 2-bit microcontrol field selects one of the following four inputs: a. BUS RD (including the Scratch Pad Register) b. D Register c. D Register shifted right d. Unibus data	The D MUX distributes 16-bit data word to: a. Instruction Register b. Scratch Pad Register c. B Register d. PS Register e. DATA display f. Internal data bus (D MUX) for basic machine and options
D Register	Four 74174 IC chips form a 16-bit temporary storage register.	Output of ALU.	Provides a 16-bit output to the D Multiplexer (D MUX) and to the Unibus data lines [BUS D (15:00)]

Table 2.1 (Cont)
PDP-11/40E Functional Components

Component	Description	Input	Output
DATA Display	Four 7380 IC chips that invert the output of the D MUX for display on the console.	16-bit output of the D MUX.	16-bit data to the console DATA indicators.
Decoding (ADRS & DATA)	Combinational logic network that decodes the Bus Address Register and generates internal control signals for addressing processor registers. Sensing is provided for stack overflow situations and zero data in the D Register.	18-bit inputs from Bus Address (BA) Register and the D Register.	Processor status (PS) Address Stack Limit Register (SLR) address (KJ11-A Option) Scratch Pad Register (REG) address Switch Register (SR) address BOVFL STOP and BOVFL signals D Register zero data.
Drivers	Three 74H04 driver IC chips provide 18 buffer gates transmitting the UPP address to the PUPP Register and to an expansion ROM.	Microprogram Pointer (UPP) output of UPP Register.	Basic Microprogram Pointer (BUPP) for application to PUPP register. Expansion Microprogram Pointer (EUPP) for an expansion ROM (KE11-E, KE11-F).
Instruction Register (INSTR REG)	Four 74175 IC chips forming a 16-bit storage register that holds the instruction.	Output of D MUX clocked the instruction fetch sequence.	Output applied to IR decode logic where it is decoded and used to control the microprogram sequence. Some bits used directly for microbranching and Scratch Pad Register selection.

Table 2.1 (Cont)
PDP-11/40E Functional Components

Component	Description	Input	Output
Instruction Register (IR) Decode	Large network of combinational logic circuits that decodes the Instruction Register instruction and generates appropriate control signals to perform the specified function.	16-bit instruction from the Instruction Register.	Generates control signals that are a function of: the operation code, instruction format, and specified register. Primary control signals are sent to the: ALU, microbranch control logic, and the BUT MUX.
JAM Microprogram Pointer (JAMUPP)	Sequential logic network consisting of flip-flops, one-shots, and decoders. This logic permits jamming an address into the UPP to modify the microprogram if certain conditions are present.	Internal control signals dependent on existing condition. Conditions causing JAMUPP are: a. bus errors b. stack overflow (red zone) c. auto restart (PWR UP) d. console switches (INIT)	Set and clear signals to UPP portion of the U WORD. Timing signals to load newly selected ROM word into the Microword Buffer (U WORD).
Processor Status (PS) Register	Four 7474 IC chips providing eight storage flip-flops to hold the processor status word. This word contains condition codes and processor priority.	Input may be either from D MUX (07:00) or may be from condition code logic.	Output may be gated onto Unibus on lines BUS D (07:00) or may be gated for processor use on lines BUS RD (07:00). Individual bits used from branch instruction decode and for microbranching.

Table 2.1 (Cont)
PDP-11/40E Functional Components

Component	Description	Input	Output
Past Microprogramming Pointer (PUPP) Register	Two 74174 IC chips providing a 9-bit storage register for keeping a history of the previous UPP address, which is the present microword address.	Loaded with the contents of the UPP Register at each system clock.	Register contents display on KM11-A Maintenance Console option when used during maintenance operation.
Register (REG) (Scratch Pad Register)	Four 3101 IC chips providing a 16 x 16 read/write facility. Basically, this represents the 16 general-purpose processor registers (referred to as the Scratch Pad Register).	Data: 16-bit input from the D MUX Control: 4-bit address input from REG ADRS input logic 2-bit read/write control from microword	Provides 16-bit data word to BUS RD buffer for transfer to one of the following: a. AIN of ALU b. BA Multiplexer c. D Multiplexer
Register Address (REG ADRS) Input	Combinational logic network used as an address multiplexer to select one of the 16 general-purpose Scratch Pad Registers for reading or writing.	There are four possible sets of inputs. One of the four is selected by the microword signals: a. IR (02:00) – 3-bit destination field from instruction register b. IR (08:06) – 3-bit source field from instruction register.	Provides address selection to the register (REG).

Table 2.1 (Cont)
PDP-11/40E Functional Components

Component	Description	Input	Output
Register Address (REG ADRS) Input (Cont)		c. RIF (03:00) – 4-bit field from microword directly d. BA (03:00) – 4-bit field from Bus Address Register Microword signals are: SRD – Selects Register Destination, IR (02:00) SRS – Selects Register Source, IR (08:06) SRI – Selects Register Immediate, RIF (03:00) SRBA – Selects Register Bus Address, BA (03:00)	
Microbranch Control (U BRANCH CONTROL)	Large network of combinational logic circuits that provide data signals for modifying the base ROM address.	Instruction Register bits IR decode signals Machine status (i.e., switches, Unibus, control flip-flops, etc.).	Data signals to the BUT MUX. These signals are used to modify the basic ROM address as a function of BUT MUX selection from the microword.

Table 2.1 (Cont)
PDP-11/40E Functional Components

Component	Description	Input	Output
Microword Control (ROM)	A read-only memory storing the KD11-A microprogram. The ROM stores 256 56-bit words. Fourteen ROM IC chips providing storage for the 256 words. Each chip stores 4 bits of the 56-bit word.	Contents of UPP Register selects the next control word to be retrieved from the ROM.	56-bit microword divided into address bits BUS U (08:00), and control bits BUS U (56:09).
Microword WORD Register (U WORD)	A 56-bit storage register consisting of type 741174 and 74174 IC chips. This register is used to buffer the output of the ROM which provides the signals defining the operation of the KD11-A data path and control.	Output of the NOT/OR gate that receives inputs from the ROM, the BUT MUX, and the EUBC for U (08:00); output of the ROM directly for U (59:09).	UPP (08:00) are the nine low-order bits of the U word which are used to select the next U word. U WORD for U (56:09) have a variety of mnemonics related to their control functions.

Table 2.1 (Cont)
PDP-11/40E Functional Components

Component	Description	Input	Output
Microprogram Pointer (UPP) Register	Five 74H74 IC chips forming an 8-bit address register. The UPP register points to the address of the next microword to be read.	Address of ROM location to be read during current machine cycle. The address loaded is a function of: a. UPF (07:00) of ROM word presently being addressed by the UPP Register. b. Basic Microbranch Control (BUBC) signals for microaddress modification (basic machine). c. Expansion Microbranch Control (EUBC) signals for microaddress modification (optional expansion).	UPP (08:00) – selects one of 256 control words stored in the ROM. It is the address portion of the U WORD Buffer noted above.

Table 2.1 (Cont)
PDP-11/40E Functional Components

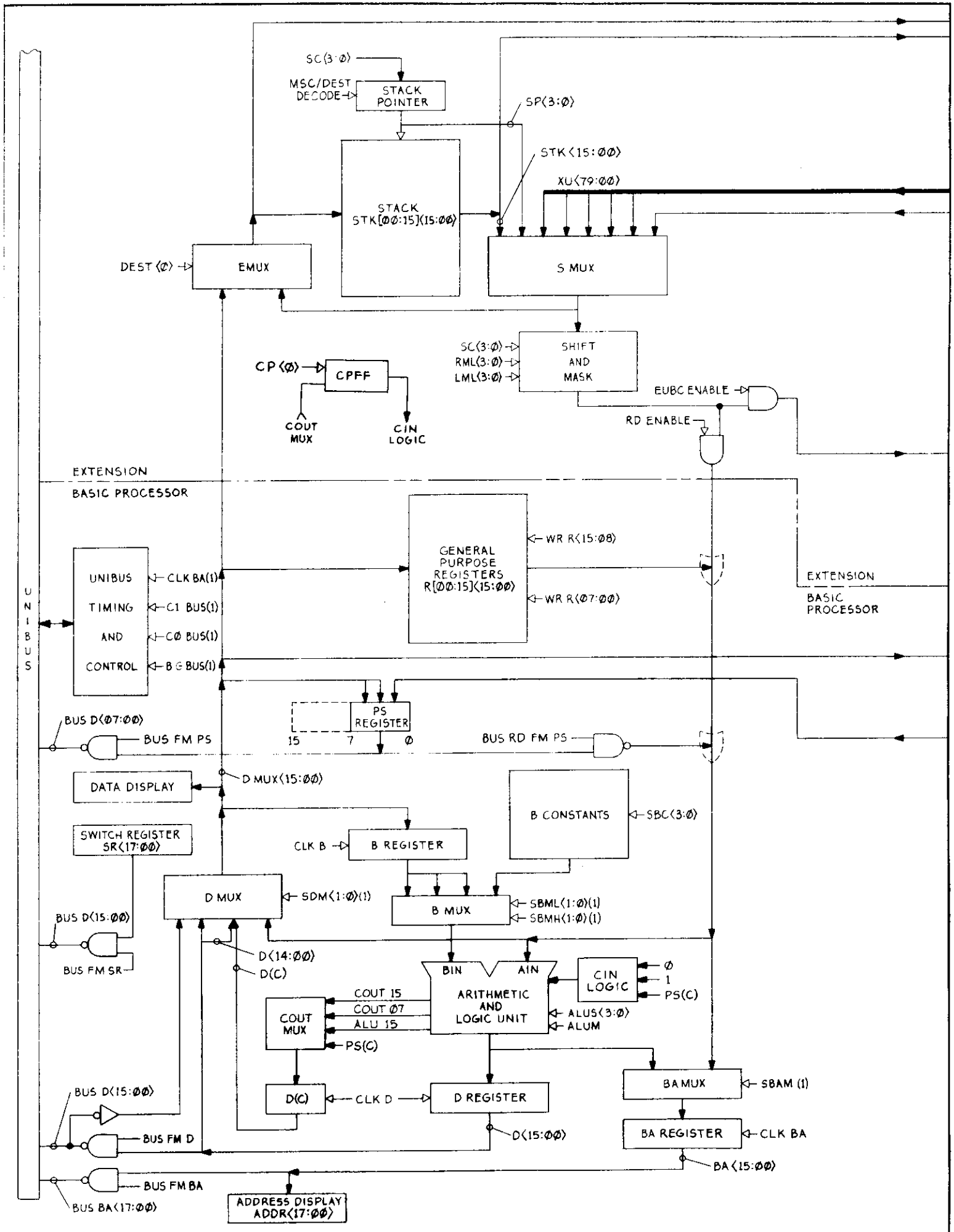
Component	Description	Input	Output
CPFF	One 74S74 D flip-flop that stores a carry bit across multi-word arithmetic operations.	Upon control of the CP bit of the XU word, is clocked from the COUT MUX output.	Upon control of the CP bit of the XU word, supplies the ALU CIN.
EMUX	Four 74158 multiplexer chips supply input to the stack and the RAM.	Upon control of the DEST field of the XU word, is clocked from either the DMUX or the SMUX output.	Supplies 16-bit data to both the stack and the RAM.
EUBC Drivers	Four 7412 chips supply input to the EUBC.	Open control of the DEST and MSC fields of the XU word, the 16-bit Mask and the 16-bit Shift outputs are received.	Supplies an 11-bit value to the EUBC.
Mask	Four IM5600 PROM chips supply a 16-bit mask for field extraction.	The RML and LML fields of the XU word.	Supplies a 16-bit mask to the RD BUS and EUBC drivers.
PROM Control Store	Four IM5600 PROM chips supply microwords to the XU word buffer.	Address received from the UPP lines.	An 80-bit microinstruction is stored in the XU word register and the U word register.
RAM Control Store	80 93415 RAM chips supply microwords to both the XU word buffer and to the SMUX.	11-bit address received from the UPP MUX. 16-bit data received from the EMUX.	80-bit data is distributed to both the XU word buffer and to the SMUX.

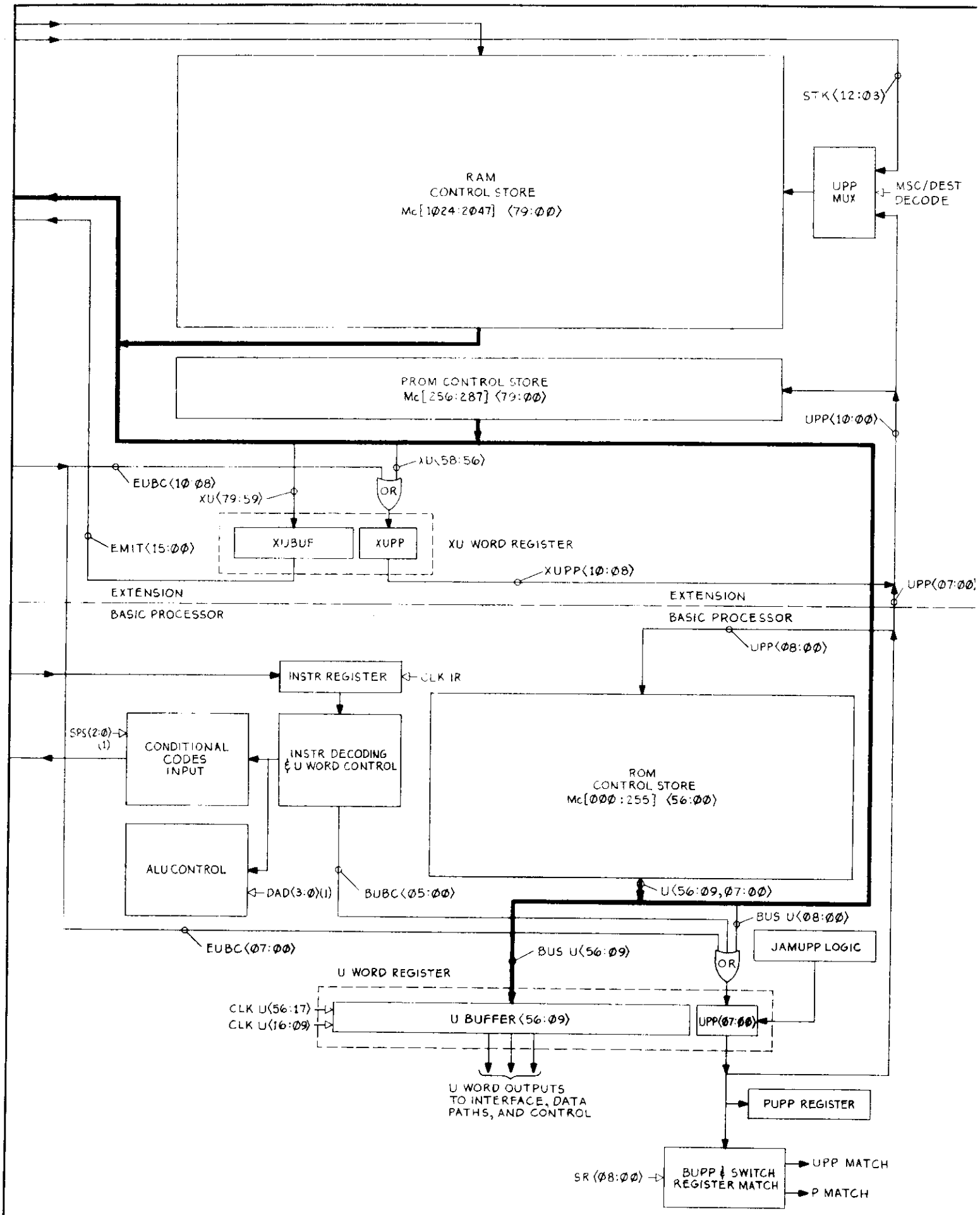
Table 2.1 (Cont)
PDP-11/40E Functional Components

Component	Description	Input	Output
RD BUS Drivers	Six 7412 chips supply input to the RD Bus.	Upon control of the DEST and MSC fields of the XU word, the 16-bit Mask and the 16-bit Shift outputs are received.	Supplies a 16-bit output to the RD BUS lines.
Shifter	Four 8243 and four 74158 chips comprise a barrel shift unit.	Upon control of the SC field of the XU word, receives the 16-bit data from the SMUX output.	Supplies 16-bit data, properly rotated, to the EUBC and RD BUS drivers.
SMUX	16 74151 multiplexer chips select the source of the extension output.	Under control of the DEST and MSC fields of the XU word, 16-bit input is selected from the Stack, the Stack Pointer, the EMIT field of the XU word, and the five 16-bit fields of the 80-bit RAM output.	The selected 16-bit value is sent to both the Shifter and to the EMUX.
Stack	Four 74S189 RAM chips provide 16 words of working memory.	Under control of the DEST and MSC fields of the XU word, a 4-bit address is received from the Stack Pointer and 16-bit data is received from the EMUX.	16-bit output is provided to both the SMUX and to the UPP MUX.
Stack Pointer	One 74191 chip provides an address for the Stack.	Under control of the DEST and MSC fields of the XU word, the 4-bit SC field of the XU word is received.	Four-bit output is supplied to the Stack address lines.

Table 2.1 (Cont)
PDP-11/40E Functional Components

Component	Description	Input	Output
UPP MUX	Three 74157 multiplexer chips select the proper RAM address.	Ten-bit input is received from both the Stack and from the UPP lines.	Ten-bit address output is supplied to the RAM.
XU Word Register	Four 74174 chips hold the 24-bit XU word as it is executed.	24-bit input from both the PROM and RAM control stores and 3-bit input from the EUBC Drivers are received.	Three-bit output to the SMUX, and various control lines are supplied.


SC(3:Ø)
MSC/DEST DECODE
STACK POINTER
SP(3:Ø)
STK(15:ØØ)
XU(79:ØØ)
STACK STK[ØØ:15](15:ØØ)
DEST(Ø)
EMUX
S MUX
SC(3:Ø)
RML(3:Ø)
LML(3:Ø)
SHIFT AND MASK
EUBC ENABLE
RD ENABLE
CP(Ø)
CPFF
COUT MUX
CIN LOGIC
EXTENSION
BASIC PROCESSOR
UNIBUS TIMING AND CONTROL
CLK BA(1)
C1 BUS(1)
CØ BUS(1)
B G BUS(1)
GENERAL PURPOSE REGISTERS R[ØØ:15](15:ØØ)
WR R(15:Ø8)
WR R(Ø7:ØØ)
EXTENSION BASIC PROCESSOR
PS REGISTER
15
7
Ø
BUS D(Ø7:ØØ)
BUS FM PS
BUS RD FM PS
DATA DISPLAY
D MUX(15:ØØ)
B CONSTANTS
SBC(3:Ø)
SWITCH REGISTER SR(17:ØØ)
CLK B
B REGISTER
D MUX
SDM(1:Ø)(1)
B MUX
SBML(1:Ø)(1)
SBMH(1:Ø)(1)
BUS D(15:ØØ)
BUS FM SR
D(14:ØØ)
D(C)
BIN
AIN
CIN LOGIC
Ø
1
PS(C)
COUT 15
COUT Ø7
ALU 15
COUT MUX
PS(C)
ARITHMETIC AND LOGIC UNIT
ALUS(3:Ø)
ALUM
BUS D(15:ØØ)
D(C)
CLK D
D REGISTER
BA MUX
SBAM (1)
BUS FM D
D(15:ØØ)
BA REGISTER
CLK BA
BA(15:ØØ)
BUS FM BA
BUS BA(17:ØØ)
ADDRESS DISPLAY ADDR(17:ØØ)
UNIBUS


RAM
CONTROL STORE
Mc[1Ø24:2Ø47] ⟨79:ØØ⟩
STK⟨12:Ø3⟩
UPP MUX
MSC/DEST DECODE
PROM CONTROL STORE
Mc[256:287] ⟨79:ØØ⟩
UPP⟨1Ø:ØØ⟩
XU⟨58:56⟩
EUBC⟨1Ø:Ø8⟩
XU⟨79:59⟩
OR
EMIT⟨15:ØØ⟩
XUBUF
XUPP
XU WORD REGISTER
XUPP⟨1Ø:Ø8⟩
UPP⟨Ø7:ØØ⟩
EXTENSION
BASIC PROCESSOR
EXTENSION
BASIC PROCESSOR
UPP⟨Ø8:ØØ⟩
INSTR REGISTER
CLK IR
SPS⟨2:Ø⟩ ⟨1⟩
CONDITIONAL CODES INPUT
INSTR DECODING & U WORD CONTROL
ROM
CONTROL STORE
Mc[ØØØ:255] ⟨56:ØØ⟩
ALU CONTROL
DAD⟨3:Ø⟩⟨1⟩
BUBC⟨Ø5:ØØ⟩
U⟨56:Ø9,Ø7:ØØ⟩
BUS U⟨Ø8:ØØ⟩
EUBC⟨Ø7:ØØ⟩
BUS U⟨56:Ø9⟩
OR
JAMUPP LOGIC
U WORD REGISTER
CLK U⟨56:17⟩
CLK U⟨16:Ø9⟩
U BUFFER⟨56:Ø9⟩
UPP⟨Ø7:ØØ⟩
U WORD OUTPUTS TO INTERFACE, DATA PATHS, AND CONTROL
PUPP REGISTER
SR⟨Ø8:ØØ⟩
BUPP & SWITCH REGISTER MATCH
UPP MATCH
P MATCH

3. MICRO/40 Assembler

3.1 Introduction

MICRO/40 is an assembler for the PDP-11/40E developed at Carnegie-Mellon University. It was written in SAIL on the PDP-10 and hence runs as a cross-assembler on the PDP-10.

3.1.1 Reserved Symbols

Since MICRO/40 is meant to be used with a particular microprocessor in mind, all of the registers and fields have predeclared names. All symbols in MICRO/40 are global and names cannot be redefined. The predeclared symbols fall into several classes.

Registers:	R, D, S, TOS, B, D/2, DSHIFT, C, BA, IR, SF, DF, EUBC, SP, UNIBUS, RAM, PS
Operators:	PLUS, +, MINUS, -, SHIFT, ↑, ←(Ascii 137), NOT, ¬(5), ~(32), ~(176), OR, ∨(37), AND, ∧(4), XOR, ⊗(26)
Pseudo Operators:	FINIS, START, END, SET, TES, CLKOFF, NOOP, TABLE, PRELOAD, LOWLIM, C., N.Z.V., N.Z.V.C.
Fields:	ALU, BUS, CLK, CLKB, CLKBA, CLKD, CLKIR, CP, DAD, DEST, EMIT, LML, MSC, PPE, RIF, RML, SBAM, SBC, SBM, SC, SCOM, SDM, SPS, SRX, UBF, WR, XUPF

3.1.2 Input Format

It is important for the user of MICRO/40 to remember that it is an assembler and has many of the limitations of an assembler. For the most part, each line of input will cause only one word of microcode to be generated. There are a two exceptions to this rule (TABLE and PRELOAD); They will be discussed in the section on pseudo operators.

A line is defined as the characters between line-feeds. Each line is read and processed individually. No distinction is made between lower case and upper case letters. Anything following an exclamation point on a line is considered a comment and it is ignored. Any line that ends with a hyphen (-) will have the next line concatenated onto it, even if this hyphen is within a comment. (The hyphen is the line continuation symbol.) Thus the following five examples are equivalent:

```
CLK = 2; RIF = 7; WR = 3; SRX = 1; SDM = 2

CLK = 2; RIF = 7; WR = 3; SRX = 1; SDM = 2;! D register to R[7]

CLK = 2; RIF = 7; -
     WR = 3; SRX = 1; SDM = 2 ! D register to R[7]

CLK = 2; RIF = 7;      .! P1 is a sufficient clock time  -
WR = 3; SRX = 1; SDM = 2;

R[7]←D
```

3.1.3 Identifiers

An identifier is a sequence of characters starting with an alphabetic and followed by an arbitrary number of alphabetics (A-Z, a-z), digits (0-9), a dot (.), or a slash (/).

3.1.4 Numbers

All numeric constants on input are considered to be octal (base 8) unless the number contains an eight, a nine, or a decimal point. Thus '102' is equal to '66.' and '11' is equal to '9'. The one exception to the rule of octal input is in field selection. See the subsection on Field Selection in the section on Assignment Statement for more information.

3.1.5 Labels

A label is an identifier followed by a colon. A line may have an arbitrary number of labels on it, and they may appear anywhere on the line.

3.1.6 Macros

Macros are defined by giving an identifier followed by := and any string not containing a dollar sign ($) or an exclamation point (!) and are followed by a dollar sign. Macros take no parameters and do strict text substitution. After a macro is expanded, the text is checked once again to see if any other macros are referenced. No delimiter is placed at the end of the expanded text, so it's possible to concatenate strings across the macro expansion.

3.1.7 When are ;'s needed

Semi-colons are normally used to separate statements within each line of microcode. The only time that they must be present is at the end of an assignment statement which is not the last statement on a line.

3.1.8 Using Common Source Files

It is often the case that the same source code should be included in more than one assembly. This is especially true for macros. (The file DEFS.MIC[N200MU00]/A is one such file that contains a number of "standard" macros.) In order to facilitate this the Require statement can be used. If the first symbol on an input line is REQUIRE (Macros cannot be used to generate this symbol), then the rest of the line is taken to be the name of a source file to replace the REQUIRE statement. After the REQUIREd file has been read, the input resumes in the original file. REQUIREs can be nested up to a depth of 12. It is not legal to have SETs and NSETs open across files.

3.2 Field Assignments

The simplest use of MICRO/40 is to turn on selected bits in each microinstruction by assigning values to each of the fields in the microword.

The syntax for this is

<field name> = <value> {;}

where the semi-colon is optional. The field names and the extended microinstruction bits they are associated with are as follows:

ALU	28:24	DEST	63:62	SBM	19:16
BUS	38:36	EMIT	79:64	SC	75:72
CLK	47:45	LML	67:64	SCOM	79:78
CLKB	41	MSC	61:59	SDM	15:14
CLKBA	39	PPE	77	SPS	31:29
CLKD	40	RIF	3:0	SRX	7:4
CLKIR	44	RML	71:68	UBF	12:8
CP	76	SBAM	13	WR	43:42
DAD	35:32	SBC	23:20	XUPF	58:48

As each field name is encountered, that field is cleared of anything that might be stored in it, and the new value is stored. No checking is done to prevent the field from being stored into twice. The fields are processed from left to right for each microinstruction.

With two exceptions the <value> is expected to be a constant. Only the EMIT and XUPF fields can get other values.

The EMIT field may store the value of a label, the address of some element in a table, or ".". If the choice is a label, then the 11-bit address of the microinstruction that contains that label will be put into the EMIT field. For a table reference the 15-bit address used to access the particular table item is put into the EMIT field. For ".", the 11-bit address of the current microinstruction goes into the EMIT field. A table reference is of the form <table id>[<n>].

The XUPF field holds the base address of the next microinstruction. If no assignment is made to the XUPF field, then a default assignment of the next microinstruction is made. It is important to remember in this context, that the XUPF field is the goto to the next microinstruction. There is no program counter for picking up the next microinstruction. A constant assigned to the XUPF field will generate a goto to that absolute location. It is usually the case that whenever the user makes an explicit assignment to the XUPF field, it is to a label.

3.3 Pseudo Operators

A number of pseudo operators have been implemented in order to relieve the programmer of some of the more mundane tasks.

3.3.1 CLKOFF

Since a field assignment clears any bits that might be in a field before storing into it, it would be difficult to turn on the low order bit of the CLK field. The CLKOFF pseudo op has the effect of turning on the low order bit so that at the end of the current microinstruction, the microprocessor clock will be turned off.

3.3.2 NOOP

This pseudo-op generates a null microinstruction; it is needed since a blank line does not generate an instruction.

3.3.3 TABLE <table name> <size>

This pseudo op makes part of the RAM a scratch pad memory. The entries in the table are considered to be 16 bits wide and are stored 4 entries to a microword. (The fifth word is really available, but it is difficult to generate the bit pattern for addressing it at run time.) The table is treated as a zero origin array. For example, assuming the declaration TABLE A 13, A[0] would be the first element, A[1] would be the second. Three microwords would be reserved for the table. (The 13 is in octal which is 11 in decimal. Four entries are used per microword, and it all fits.) It is important to use the []'s whenever a reference to a table is made.

3.3.4 PRELOAD <table name> <value list>

This pseudo op is very much like the TABLE pseudo op. The sole difference is that the items in <value list> are preloaded into the table. <value list> consists of a list of compile time constants optionally separated by commas. Since table entries are considered compile time constants, the user can put the addresses of table entries into a table.

3.3.5 FINIS

Finis indicates the end of the input; any remaining text in the file will be ignored. If their is no FINIS in the file, then a continuable error will be generated.

3.3.6 . = <value>

The syntax for assigning to '.' is very much like that for Fields. <value> must lie in the range 2000 to 3777. The function of this pseudo operator is to force MICRO/40 to locate the associated microinstruction at some particular word in the RAM. The assignment to '.' may occur anywhere on the input line as in the assignment to Fields.

Lines that have no explicit address are assigned addresses by MICRO/40. MICRO/40 handles the XUPF field to make sure that the XUPF field of any instruction which is supposed to precede any particular instruction points to the correct location.

3.3.7 LOWLIM = <value>

This pseudo op sets the lowest address to which microcode will be assembled. <value> must lie between 2000 and 3777. The default value of LOWLIM is 2000 and is used if this operation is not specified.

3.3.8 SET ... TES

A PDP11/40 has no program counter for the microcode. In order to do conditional microbranching, the UBF field causes certain bits in the next address (of the succeeding instruction) to be ORed in. In the 11/40E, one of the extensions enables values out of the EMIT field or values from the Stack to be ORed into that same address. It is important that the addresses to which the microbranches expect to go be the same, except in those bits that might be turned on. It would be possible (by using ".=<value>") to put these instructions where they could be used, but the SET...TES combination usually obviates the need for explicit space allocation.

The syntax is to have SET on a line by itself, the "controlled" microinstructions , and then TES on a line by itself. MICRO/40 counts the number of controlled microinstructions, finds contiguous words in the RAM on a 2^n word boundry and puts the controlled microinstructions there.

It is illegal to assign a word controlled by a SET ... TES group to a specific location through the use of the .=<value> construct. The default goto is to the instruction following the the SET ... TES.

3.3.9 START ... END

It is often desired to do more than one microinstruction on a fork of a branch. To do this, the user may utilize the START ... END construct. Inside the SET ... TES, put START on a line by itself. The first instruction will be assembled into the block of instructions associated with the SET ... TES. The following instruction will not be in that same block, but the goto of that first instruction will be to the second instruction as expected. The remaining instructions in the compound are treated similarly. The last instruction of the compound has a default goto to the instruction after the SET ... TES.

It is legal to nest SET ... TES groups. Because of the way the hardware is constructed, it does not make sense to have a second SET start while in a top level instruction of the first SET. A SET pseudo-op may appear anytime after the first instruction in the START ... END pair.

3.3.10 C., N.Z.V., N.Z.V.C.

These pseudo operators set the SPS field so that the carry bit; the negative, zero, and overflow bits; or the negative, zero, overflow, and carry bits of the Processor Status Word will reflect the current conditions. The CLK field is also modified to insure proper timing.

3.4 The Assignment Statement

3.4.1 Syntax

The basic form for the Assignment Statement is <register list> ← <expression>

<register list> is a list of registers separated by commas.

<expression> is of the form
<A-op> | <A-op> or <B-op> | <A-op> or not <B-op> | minus 1 |
<A-op> plus <A-op> and not <B-op> |
(<A-op> or <B-op>) plus <A-op> and not <B-op> |
<A-op> minus <B-op> minus 1 | <A-op> and not <B-op> minus 1 |
<A-op> plus <A-op> and <B-op> | <A-op> plus <B-op> |
(<A-op> or not <B-op>) plus <A-op> and <B-op> |
<A-op> and <B-op> minus 1 |
<A-op> plus <A-op> | (<A-op> or <B-op>) plus <A-op> |
(<A-op> or not <B-op>) plus <A-op> |

<A-op> minus 1 | <A-op> plus 1 | (<A-op> or <B-op>) plus 1 | 0 |
<A-op> plus <A-op> and not <B-op> plus 1 |
(<A-op> or <B-op>) plus <A-op> and not <B-op> plus 1 |
<A-op> minus <B-op> | <A-op> and not <B-op> |
<A-op> plus <A-op> and <B-op> plus 1 |
(<A-op> or not <B-op>) plus <A-op> and <B-op> plus 1 |
<A-op> and <B-op> | <A-op> plus <A-op> plus 1 |
(<A-op> or <B-op>) plus <A-op> plus 1 |
(<A-op> or not <B-op>) plus <A-op> plus 1 | not <A-op> |
not (<A-op> or <B-op>) |
not <A-op> and <B-op> | not (<A-op> and <B-op>) |
not <B-op> | <A-op> xor <B-op> |
<A-op> and not <B-op> | not <A-op> or <B-op> |
not (<A-op> xor <B-op>) |
<B-op> | <A-op> and <B-op> | <A-op> or not <B-op> |
<A-op> or <B-op> |
UNIBUS | D | D/2

<A-op> may be anything that can be placed on the RD lines. One possible <A-op> is a general register of the form R[<number>], R[SF], R[DF], or R[BA]. The other standard <A-op> is PS, the processor status word. In addition, the extension can gate the current top of stack, TOS (or S, if the stack is to be popped); the extension stack pointer, SP; the emit field of the form <octal constant>, <label>, or <table address>; or a 16-bit word from the RAM of the form RAM[TOS] or RAM[S]. If multiple occurrences of an <A-op> are needed, they should all be identically specified within the assignment statement. This somewhat arbitrary list of possible expressions is a direct consequence of using the 74181 integrated circuit to implement the ALU. <A-op>'s can be optionally followed by <field specifier>, <shift factor>, or <field specifier><shift factor>. A <field specifier> is of the form "<l:r>" or "<l>". A <shift factor> is of the form "SHIFT n". "l", "r" and "n" are decimal (not octal !!!) constants.

In any <expression> the following equivalences hold:

PLUS ↔ +
MINUS ↔ -
SHIFT ↔ ↑

If one uses SOS and thinks in terms of the Stanford character set then the following equivalences hold:

or ↔ ∨ = ?8
and ↔ ∧ = ?$
xor ↔ ⊕ = ?1
not ↔ ~ = ?4 ↔ ¬ = ?% ↔ ~ = ?3

<B-op> is of the form B (meaning the B-register) or B.<high selector><low selector> where <high selector> is one of H, E, L or C and <low selector> is one of L, H or C. B can also be one of the B-constants: 0, 1, 2, 177570, 17, 77, 250 or 4.

3.4.2 Semantics

The crucial piece of information to remember when writing assignment statements is that each action that is desired must be specified explicitly. The user might hope that

R[7]←R[7] + 2

would increase the value of R[7] by 2 and as a side effect leave the new value in the D register. Unfortunately, MICRO/40 is not that intelligent. It is necessary to write

D←R[7] + 2; R[7]←D

to accomplish the statement. Note that

R[7]←D; D←R[7] + 2

is identical in its affects to writing the statements in the other order.

The clock field has a bit set if necessary to accomplish the clocking of a register in <register list>. For the previous example, a P3 clock cycle would be generated since both the D-register (which takes a P2 pulse) and one of the general-registers (which get clocked on either a P1 or P3 pulse) are being clocked. A P3 clock cycle generates both a P2 and P3 pulse.

Whenever the symbol S is used on the left hand side of an assignment statement, the stack is pushed before the value is stored. When, TOS is used, the new value is written over the current top of stack. On the right hand side of an assignment statement, S results in the value being read out of the stack and the stack being popped. TOS results in the value of the top of stack being read out.

Values that are coming onto the RD lines from the extension or into the EUBC lines may be shifted or masked. The default mask is for the entire word to be put out and the default shift is zero. Values that are only masked are shifted (by MICRO/40) to the right so that they are right justified. Values that are only shifted are rotated to the left by the number of bits specified. Values that are both masked and shifted act as if the shift count is applied after the field has been right justified. It is illegal to specify a shift-mask combination that results in the final value not being contiguous. All shift and mask amounts are specified in decimal. For example, "D←S<11:3>↑2;" would place into the D register 9 of the bits, shifted 1 to the right, from the result of popping the top value off the stack.

3.5 Examples

The following are some macros that are defined in DEFS.MIC[N200MU00] and have been found to be of general use.

```
! standard definitions for micro -- 11 October 1974
!                                  rev: 19 November 1974
!                                  rev:  7 December 1974
!                                  rev: 11 June 1975
!
r0 := r[0]$;    r1 := r[1]$;    r2 := r[2]$;    r3 := r[3]$
r4 := r[4]$;    r5 := r[5]$;    r6 := r[6]$;    r7 := r[7]$
r10 := r[10]$; r11 := r[11]$; r12 := r[12]$; r13 := r[13]$
r14 := r[14]$; r15 := r[15]$; r16 := r[16]$; r17 := r[17]$
rsp := r[6]$;   rpc := r[7]$;   rdf := r[df]$; rsf := r[sf]$
temp := r[10]$; rsrc := r[11]$; rdst := r[12]$
```

```
rir  := r[13]$; vect := r[14]$; temc := r[15]$
spus := r[16]$; adrsc := r[17]$;       rba := r[ba]$
dati := bus=1$;        dato := bus=5$
datip := bus=3$;       datob := bus=7$
p1 := clk=2$;  p2 := clk=4$;  p3 := clk=6$
exit := xupf = 16$  ! return to rom
begin := beg: .=2000;$
goto := xupf =$;       case := eubc←$; popst := d←s$
but := ubf =$; skipzero := ubf = 12$ ! skip on d = 0
return := eubc←s$;     endproc := xupf=0$
smod := 11:9$; dmod := 5:3$;  prop := cp=1$
! end of macros
```

The following examples assume that the prior macros have been defined.

3.5.1 Add the numbers from 1 to 10

```
entry:  D←0; B,R[0]←D
        D←10; R[0]←D
L:      D←R[0]+B; B←D;
        D←177777+B; B←D; skipzero
        noop
        set
           goto L
           noop
        tes
        <whatever comes next>
```

It is not difficult to optimize the above code, but it was written for clarity. An optimized version could be

```
entry:  D←0; R[0]←D                          ! clear R[0], our accumulator
        B←10                                 ! clock B from the EMIT field
        set
LOOP:       D←177777+B; B←D; skipzero        ! decrement B; note A op B format
          <whatever comes next>; goto NEXT   ! since B was zero, break the loop
        tes
        D←R[0]+B; R[0]←D; goto LOOP          ! add B to R[0]
NEXT:   <second instruction after loop>
```

3.5.2 360 instruction decoder

```
        IC   := R[10] $                    ! 360 instruction counter
        BASE := R[11] $                    ! base of register
        IREG := R[12] $                    ! instruction register

DECODE: BA←IC; dati                        ! fetch 1st halfword of instruction
        D←IC+2; IC←D; clkoff               ! increment 360 instruction counter
        S,IREG←unibus                      ! get 1st halfword to IREG, STACK
        case TOS<15:12>                    ! decode 1st hex digit of IREG
        B←BASE
        set
          goto BRSTAT                      ! Branch and Status Op-codes
          start                            ! fixed-point RR in a little detail
            BA←TOS<7:4>↑2 + B; dati        ! fetch 1st half of R1
            D←2+B; B←D; clkoff
            R[0]←unibus; BA←TOS<7:4>↑2+B dati
                                           ! fetch 2nd halfword of R1
                .
                .
                .
          end
          goto LONGRR                      ! long real RR op-codes
          goto REALRR                      ! real RR op-codes

          goto BYTRX                       ! byte RX branching
          goto WDRX                        ! word RX
          goto LONGRX                      ! long RX
          goto REALRX                      ! real RX

          goto RSS18                       ! further decode if necessary
          goto RSS19                       ! further decode if necessary
          goto ILLEGAL
          goto ILLEGAL

          goto ILLEGAL
          goto LOGSS                       ! logical SS
          goto ILLEGAL
          goto DECSS                       ! decimal SS
        tes
```

3.6 Using MICRO

3.6.1 Running MICRO

To assemble a MICRO/40 routine, type R MICRO to the TOPS-10 monitor. It will prompt with a "*". MICRO/40 accepts the standard DEC command string format.

<object file>,<list file>←<source file>/<switch>/<switch>

If there is no comma before the left-arrow, then no listing file will be generated. If no file is specified before the comma, then the object file will not be generated. If MICRO/40 is run at CMU, whenever an object file is created, MACX11 is run automatically to generate an .OBJ file from the .P11 file that MICRO/40 writes. The default extensions are .P11 for the <object file>, .LST for the <list file> and .MIC for the <source file>.

The legal switches are DIAGNOSTICS, SIMULATE, and NOMACX11. (Unique abbreviations are accepted.) SIMULATE causes the simulator to be invoked after the assembly is complete. DIAGNOSTICS writes out special <object file>s that can be linked in with the hardware diagnostics for the PDP11/40E. NOMACX11 suppresses the running of that program.

3.6.2 Errors

Errors are either continuable or non-continuable. Non-continuable errors are indicated by having MICRO/40 output the text line in which it found the error followed by the error message, a <crlf>, and finally a ?. These errors are such that any attempt to continue would cause MICRO/40 to bomb out.

Continuable errors have the ? replaced with an ↑. (The entire user-error interface is the one provided by SAIL in which MICRO/40 is written.) A carriage-return to the error prompt will cause the assembler to continue. A bare line-feed will cause MICRO/40 to continue automatically. This means that error messages will be typed out, but the assembler will not stop.

If E is typed as a response to the error prompt, then LINED will be entered (pointing to the error line if the file has line numbers on it).

4. Microprocessor Simulator

4.1 Summary

The 11/40E simulator allows interactive testing on the PDP-10 of programs written in MICRO/40. It provides facilities for tracing execution, setting breakpoints, examining and changing the contents of registers during execution, and timing sections of the program.

The simulator is invoked through the MICRO/40 assembler by use of the S switch in the command line. After assembling the program, the assembler will call the simulator, putting it in communication with the user.

When the simulator is first entered, it goes into command mode -- indicated by its prompting with ">". It also goes into command mode whenever execution is halted (because of errors, breakpoints, etc.). The commands listed below are always legal, regardless of why command mode is entered or what commands have been given previously. If a command is given which conflicts with a previous command the simulator obeys the more recent command.

The simulator leaves command mode and begins execution of the program as an effect of the G or S command. It can return to command mode for any of several reasons, including the encountering of certain errors, the actions implied by some of the commands listed below, and the receipt of a halt command from the console (at any time during execution, the user may type H or h which will cause the simulator to halt after the current microinstruction).

4.2 Commands

Commands are written one per line in the format given below. The simulator performs a very simple scan looking for "=" and "," which delimit the strings representing names, numbers, etc. Some of these strings are processed by the simulator, but most are sent back to the assembler which makes it possible to have anything which the assembler would interpret as a register name (say), including macros defined in the program, appear in a command line where a register name is needed.

In the commands below, four possibilities, <number>, <name>, <register>, and <label>, are shown for the strings. <number> means an integer in the range -32768 to 65535 (numbers greater than 32767 make sense if the 16 bit words are assumed not to contain a sign bit) and which are assumed to be written in octal unless they contain an 8, 9, or trailing decimal point in which case they are assumed to be decimal. <name> means anything that will be recognized by the assembler as the name of a register or label. <register> and <label> are similar except the simulator has some expectation about what the string should represent. Whenever a <label> appears, a <number> representing the instruction address may appear instead.

In showing the syntax of the command lines, two meta symbols have been used. Things enclosed in "[]" are optional (they may appear one or zero times). Things enclosed in "{}" may appear zero or more times.

G[<label>]

Go. Continue execution of the program. If a label is given, execution begins at that location; if not, execution begins where it was last interrupted. Giving a G command without a label before any instruction has been executed causes execution to begin at location 2000 (octal).

= <register> {, <register>}

Get. Print the contents of the registers. Values are printed in octal.

T <name> {, <name>}

Trace. Registers are traced by printing the register's name, as typed to the T command, and its new contents after each change. Instructions are traced by printing the label, as typed to the T command, each time that instruction is executed. Registers and instructions with more than one name can be traced under only one name at a time. If an attempt is made to trace one under two different names, only the last name given will be used. There is a limit to the total number of registers and instructions which may be traced simultaneously, but this limit is large and unlikely to cause any problems.

B <name> {, <name>}

Break. The B command is a variant of the T command which causes a break (i.e., causes a halt and a return to command mode) after printing the trace information. The instruction broken on (or in which a register being broken on is changed) will be completed before the break occurs. Since this is a variant of the T command the same restrictions about names and number of registers or instructions broken on apply. In addition, it is not possible to both trace and break on a register or instruction simultaneously. If both B and T commands have been issued for the same register or

instruction, the last command issued will be the one obeyed (and if different names were given, the last given will be used).

← <register> = <number>

Set. Set the contents of the register to the given value. The change is made immediately, causing the old contents to be lost.

R [<name> {, <name>}]

Remove. Stop tracing or breaking on the registers and instructions listed. If no names are given, all tracing and breaking will be stopped. It is not necessary that the same names be given for the registers and instructions that were given in the T or B commands to be removed -- the simulator responds to the registers and instructions, not their names.

S [<label>] [, <number>]

Step. Step through the program. If a label is given, the stepping begins at that location; if not, stepping begins where the program was last halted. If no label is given and no instructions have been executed, stepping begins at location 2000. If a number is given, the program is stepped that many times then halted. If no number is given, 1 is assumed. While stepping, the address (not the label) of each instruction will be printed out before the instruction is executed.

C

Clock. Print the total simulated time and the time since the last C command. The time is in nanoseconds, printed in decimal. The time for each instruction is computed by:

$$T = Te + Tm + Tr$$

where

Te = execute time
Tm = memory access time
Tr = regenerate time.

Execute time is: if the instruction is a P1, 140 nsec; if the instruction is a P2, 200 nsec; if the instruction is a P3, 300 nsec. Memory access time is: if the instruction performs a CLKOFF and it is less than 500 nsec since the memory access began, sufficient time to make the difference 500 nsec, otherwise 0. Regenerate time is: if the instruction performs a CLKOFF and the time between the last two memory accesses is less than 900 nsec, sufficient time to make the difference 900 nsec, otherwise 0.

E

Exit. Return control to the assembler.

L <file>

Load. Load the 11/40 core from the file named. Each line of the file will contain the data to be loaded into one 16 bit word in the format:

[<number> :] <number>

The second number is the datum to be loaded. The first number, if present, is the word address in core to be loaded; this number must be even (it is a word address) and in the range 0 to 1022, for a total of 1024 bytes of simulated core. If no address is provided, the address loaded is the last address plus 2; if no address is provided for the first line of the file, that word is loaded into location 0. Loading terminates on end of file. <file> may be any file or device specification acceptable to Sail, including TTY:. If input is through TTY:, everything typed after giving the command is treated as input and should be in the above format; end of file (↑Z on TTY:) terminates input and returns to command mode.

D

DDT. Call DDT. This command works only if DDT has been loaded and the user is one of the system maintainers.

4.3 Inconsistencies

The simulator doesn't attempt to model the entire processor. Omitted are some things related to particular pieces of hardware (the console, peripherals, and memory management) and the functions which are useful only in emulating a PDP-11.

The omissions are:

1. BUS=2 and BUS=6 are noops.

2. All DAD except DAD=10 (and the corresponding part of DAD=11) are noops.

3. The SBC which test conditions always return zero for the condition.

4. All UBF except UBF=12 and UBF=17 always return zero.

5. Programming Techniques

Since the 11/40E microprocessor is more difficult to program and use than a conventional macroprocessor, the arguments by Dijkstra in ["A Constructive Approach to the Problem of Program Correctness", BIT, Jul 68] apply even more strongly. The programmer must not turn out sloppy code and then hope to debug it; he must rather design his algorithm carefully and then code it with a solid understanding of both his algorithm and the machine. There are two particular reasons for this caveat:

1 the 11/40E is a much more complex machine than that seen by the machine language programmer on a conventional processor. It will try to execute any bit pattern and will detect very few nonsensical microinstructions. The variety of actions that may occur from program errors is frightening. Just as the subspace of meaningful programs is sparse in the space of valid assembly programs, so the subspace of meaningful microinstructions is sparse in the space of valid microinstructions. Many program errors will act in unpredictable ways that may vary from day to day, depending on the relative speeds of microprocessor components.

2 debugging aids on the 11/40E are very rare. A microroutine that does jam or hang will not trap to a diagnostic routine that can save the microprocessor state.

Having given this warning, we attempt to outline some programming styles we wish to encourage and some of the more subtle 'features' of the machine that need to be remembered when coding.

5.1 Timing

An 11/40E microinstruction takes one of three cycle times between 140 nsec and 300 nsec. While the MICRO/40 assembler attempts to set the proper timing for each microinstruction, the programmer should be aware of the timing requirements of each operation in his microinstruction, so that he can judge the performance of alternative microcode sequences.

The shortest cycle is called 'p1' and ends with a pulse, called the 'p1 pulse', 140 nsec after the start of the microinstruction. This duration is sufficient for most transfers that don't involve the ALU or pushing the extension stack pointer.

The second cycle time is called 'p2' and ends with the 'p2 pulse' at 200 nsec. This duration is required for the ALU output to become stable and for the extension stack pointer to be incremented prior to a push onto the stack. It is also required for simple transfers over a long distance, e.g. moving a general register to the stack.

The longest cycle is called 'p3' and contains both a 'p2 pulse' at 200 nsec and a final 'p3 pulse' at 300 nsec. This timing is used when the D register must be clocked from the ALU, then sent up the DMUX bus.

All the standard registers on the DMUX bus, B, IR, PS, and R[0:17], may be loaded only on a p1 or p3 pulse. Hence a transfer from the EMIT field of the extension to a register, which is too long for a p1 cycle, must take the full 300 nsec of a p3 cycle.

5.2 B Constants

In the standard 11/40 microprocessor, the only constants available to the programmer were a handful of constants essential to PDP-11 emulation. The extended 11/40E provides the programmer with arbitrary constants via the EMIT field, but these constants must be A inputs to the ALU. Careful use of the B-constants can expedite many microroutines that would otherwise have to copy one operand to the B register before using the ALU.

5.3 Unibus Control

As in the PDP-11 architecture, memory and peripheral registers are accessed in a uniform fashion. For the microprogrammer, however, this accessing is more explicit; the details of bus control and timing combine to present an increased opportunity for both efficiency and error. As background for this subsection, see the description of Unibus conventions in [Digital 73].

In order to read a Unibus device, the bus address must be clocked to the BA register as, or before, the DATI bus control code is asserted. The microprocessor will correctly handle the relative timing of the address and DATI assertions and will properly wait if the bus is already busy. Succeeding microinstructions may continue processing, provided they do not modify the BA register or assert a second bus control code. Prior to the completion of the Unibus access, the microprocessor should pause via the CLKOFF construct. Upon completion of the access, the data will be present on the Unibus and the clock will be restarted. This Unibus data can then be gated through the DMUX to the microprocessor registers. MSYN is dropped only upon a p1 or p3 cycle following this CLKOFF, so the instruction after the CLKOFF should be a p1 or p3, and should pull the data off the Unibus immediately. A second Unibus request can then be made, but the cycle time of the memory may limit the actual rate at which they are performed. The ability to overlap processing with Unibus access provides the microprogrammer with a great opportunity for efficiency. With some care, he can keep the microprocessor totally Unibus-bound while doing some processing in addition to Unibus control.

Writing onto the Unibus is done similarly. As before, the bus address must be clocked to the BA register. The datum to be written must also be clocked to the D register and the DATO bus control code must be asserted. As with the DATI access, processing may be overlapped with Unibus access, but now both the BA register and the D register must be kept constant. This makes the overlap feature less valuable than with DATI access. As before the instruction following the CLKOFF should be p1 or p3 so that MSYN may be dropped.

The DATIP Unibus cycle is used to implement a read-modify-write cycle on the Unibus. Programming is as for a DATI; a normal DATO follows. Examples of the three cycles discussed above are shown below:

```
pop:    BA←R[6]; DATI                    ! pop the stack to r2
        D←R[6]+2; R[6]←D; CLKOFF
        R[2]←UNIBUS
```

a. DATI Example

```
push:   D,BA←R[6]-2; R[6]←D              ! push r2
        D←R[2]; DATO; CLKOFF
        <... (p1 or p3)>
```

b. DATO Example

```
incr:   BA←R[6]; DATIP; CLKOFF           ! increment the
        R[10]←UNIBUS                     ! top of stack
        D←R[10]+1; DATO; CLKOFF
        <... (p1 or p3)>
```

c. DATIP-DATO Example

Unibus control for byte addressing is similar to that for whole words. For reading a byte, the DATI bus control is used as for words. The whole word returns and can be brought through the DMUX lines. To select the proper byte, the address must be tested; if even, the byte is in the lower half of the word -- if odd then it's in the upper half. The B register is useful for aligning the byte. Either the EUBC←TOS<0> construct, where the address is in the TOS, or the BUT 35 (see the ROM microcode) are useful for detecting an odd address.

On writes, the DATOB bus control is used. Programming is similar to that with DATO with words, but the datum being written must be in the proper byte of the D register. Thus, if the address may be either even or odd, the datum must be duplicated in both bytes of the D register. The B register facilitates this.

Finally, whenever an odd address is clocked into the BA register or whenever the DATOB bus control is asserted, two conditions must be met: The IR must contain a valid byte instruction (!) and bit 0 of the DAD field must be set. Otherwise, a jam will occur and flow of control will be lost.

5.4 Flow of Control

5.4.1 Introduction

Understanding the flow of control mechanisms on the 11/40E requires review of the basic fetch/execute cycle. Whenever a microinstruction is clocked from the control store to the microinstruction buffer, the XUPF field is modified by ORing in the eleven bits of the EUBC and the six bits of the BUT lines. This modified XUPF is used immediately as the address of the next microinstruction; fetch of this next microinstruction overlaps the execution of the current microinstruction. Since control store access time is shorter than any microinstruction execution time, this scheme eliminates any delay due to microinstruction fetch. It does, however, make it impossible for a microinstruction to influence the address of its successor. Instead, microbranching is performed by setting the EUBC/BUT lines so that the address of the successor of the next microinstruction is altered.

Several points can now be made. First, the delayed branching of this mechanism is unnatural for most programmers and requires great care. Even when the programmer understands it, the feature makes a microprogram difficult to modify due to the interdependence of the microinstructions in a branching sequence. Second, since the branching is always due to ORing bits into the XUPF field, care must be taken to leave the proper bits of the base address clear for this ORing to have the right effect. This is accomplished in MICRO/40 via the SET...TES construct, which places a set of 2↑n microinstructions in contiguous locations whose addresses begin at a multiple of 2↑n. Finally, it should be noted that the mechanism offers a very wide branch with the same speed as a two-way branch.

5.4.2 BUT

The 11/40E provides two mechanisms for setting the bits to be ORed into the XUPF. The first is provided by the standard 11/40 and is called BUT, for Branch Micro Test. This mechanism consists of about thirty different tests invoked by the different codes in the UBF field of the microinstruction. These tests detect processor conditions and set the appropriate bits on the BUT lines. Unfortunately, most of the conditions detected are specifically oriented toward PDP-11 emulation, especially instruction register and console conditions. Two of these deserve notice, however. The BUT 12 tests the D register and ORs a one into the BUT lines if the D register is identically zero. If the microinstruction is p1 or p2, then the value of the D register at the beginning of the instruction is tested. If, however, the timing is p3, then the value clocked into the D register at p2 is tested. The second important test is BUT 16, which detects any conditions that would cause an interrupt at the next PDP-11 instruction fetch (e.g. an I/O interrupt or the halt switch), and ORs in a one if such a condition is detected. This BUT is normally used just prior to the last microinstruction executed in the RAM. The last microinstruction has an XUPF field of 16. Control is then passed to either 16 for the next PDP-11 instruction fetch or to 17 for the handling of the interrupt.

5.4.3 EUBC←...

The second branching mechanism centers on the field extraction unit of the extension. Any value that can be brought down the RD bus from the extension can instead be brought down the EUBC lines. In particular, an arbitrary contiguous field can be pulled from the word at the top of the extension stack; the EMIT field can also be brought down.

5.4.4 if statements

The simplest use of these mechanisms is to implement a simple if statement. We begin with two uses of the BUT 12, showing the timing quirk mentioned above.

```
D←R[0]
D←R[1]; BUT 12
B←D
set
  <...>              ! r0 non-zero
  <...>              ! r0 = 0
tes
```

a. Example of BUT 12 if on p2

```
D←R[0]
D←R[1]; B←D; BUT 12
NOOP
set
  <...>              ! r1 non-zero
  <...>              ! r1 = 0
tes
```

b. Example of BUT 12 if on p3

```
TOS←R[0]
EUBC←TOS<15>
NOOP
set
  <...>              ! r0 non-negative
  <...>              ! r0 negative
tes
```

c. Example of EUBC←... if

5.4.5 case statements

Multiway branches are vital to efficient emulation, yet they offer little conceptual difficulty over that of the if construct. Refer to Appendix B for examples of case constructs based on BUTs, especially for op-code and address-mode decoding. Refer also to Appendix A for examples of address-mode decoding using the EUBC←... mechanism of the extension. We present here a three-way case construct, which branches differently for positive, negative, and zero values of R[0].

```
D←R[0]; TOS←D
BUT 12; EUBC←TOS<15>↑1
NOOP
set
  <...>          ! positive (non-neg; non-zero)
  <...>          ! zero     (non-neg;     zero)
  <...>          ! negative (    neg; non-zero)
  <...>          ! impossible (  neg;     zero)
tes
```

d. Example of FORTRAN II style if construct

5.4.6 Subroutine Call/Return

Conventional recursive subroutines are easily implemented in the 11/40E. Upon call, the return address is pushed onto the extension stack and the XUPF field is set to the subroutine address. Just prior to return, this stack is popped onto the EUBC. The final instruction has a clear XUPF field so that the popped value can form the effective address of the next microinstruction.

```
caller: S←retadd; goto subr
retadd: <...>
   .
   .
   .
subr:  <...>
   .
   .
   .
       <...>; EUBC←S
       <...>; XUPF=0
```

e. Sketch of Call/Return Mechanism

In experience at CMU, parameter passing has been ad hoc, but the style of call/return shown above has been a very valuable model.

5.4.7 Emulator Flow of Control

The most important exception to the conventional subroutine linkage occurs in emulator writing, either in extending the PDP-11 instruction set or in emulating foreign instruction sets. The technique is simple: keep a copy of the instruction being emulated at the top of the extension stack. Then direct the flow of control from instruction fetch to operand fetch to execution routines by pulling various fields from the top of stack onto the EUBC.

5.5 RD Bus Details

The RD bus has three potential sources: the general registers, the processor status word, and the extension. Each of the three can independently gate a word onto the RD bus. Usually two sources gated onto the RD bus would be an error. Suppose, for example, we are to set the stack pointer (R[6]) to 400(octal). A naive programmer might write:

```
D←400; R[6]←D
```

Now the MICRO/40 assembler will allow this, but since R[6] is being clocked, its old value will be gated onto the RD bus. Thus the new value of R[6] is its old value with bit 8 set. To perform the intended task, the sequence would have to be split into two instructions. This feature can, however, be used to produce a positive effect, as when a table lookup is to be performed into a table of fixed location, indexed by a register. The sequence is:

```
BA←R[0]+R[0]; DEST=0; MSC=1; EMIT=tablebase; DATI
```

Here tablebase is the word address of the fixed table and has enough zeroes in its low-order part that r0+tablebase=r0vtablebase. The effective value on the RD bus is R[0]vtablebase, which gives the word address of the table entry, provided that tablebase is properly chosen. The value clocked onto the BA register is double this value, or the proper byte address. Were the ORing feature of the RD bus not used, the sequence would be:

```
D←R[0]+R[0]; B←D
BA←tablebase+B; DATI ! where tablebase is now the byte address
```

This would add 300 nsec and an instruction to the sequence and destroy the old contents of the B register.

5.6 The Last Word

Whenever the XUPF field of a microinstruction buffer is less than 256, the extension is turned off. Consequently, the last microinstruction executed in the RAM, which exits to the ROM, may not use the extension hardware.

Appendix A. Microinstruction Format

A.1 The Word

U<56:09,07:00> = The location in the 11/40 56 bit microinstruction.
XU<79:00> = The location as rearranged in the 80 bit writable microinstruction.
G<4:0><15:00> = The 5 groups for writing into the writable microinstruction from the PDP 11.

Name / Location	Description
BUS<2:0>	
U<47:45>	C1 C0 BG
XU<38:36>	0 0 0 -
G2<6:4>	0 0 1 - DATI
	0 1 0 - AWBBY ←1 (await BUS BUSY)
	0 1 1 - DATIP
	1 0 0 -
	1 0 1 - DATO
	1 1 0 - restart on perif release
	1 1 1 - DATOB
CLKB	Allows clocking DMUX <15:00> into the B REGISTER.
U<50>	0 - NO-OP
XU<41>	1 - B REGISTER ← DMUX output
G2<9>	Timing: P1 ∨ P3
CLKBA	Allows clocking the BUS ADDRESS REGISTER.
U<48>	0 - NO-OP
XU<39>	1 - BA REGISTER ← BAMUX output
G2<7>	Timing: P1 ∨ P2
CLKD	Allows clocking the ALU into the D REGISTER
U<49>	0 - NO-OP
XU<40>	1 - D REGISTER ← ALU output
G2<8>	Timing: P2
CLKIR	Allows clocking the UNIBUS DATA into the INSTRUCTION REGISTER.
U<53>	0 - NO-OP
XU<44>	1 - INSTRUCTION REGISTER ← DMUX output
G2<12>	Timing: P1 ∨ P3
CLKOFF	
U<54>	0 - NO-OP
XU<45>	1 - turn off processor clock
G2<13>	(See DAD table for exceptions.)

CLK<1:0>
U<56:55>
XU<47:46>
G2<15:14>

Processor clock length control
0 0 - P1 140 ns.
0 1 - P1 140 ns.
1 0 - P2 200 ns.
1 1 - P3 300 ns. (also gives P2 pulse)

CP
XU<76>
G4<12>

Carry Propagate into and out of ALU. (See also SCOM)
0 - NO-OP
1 - ALU CIN00 ← CPFF;
CPFF ← COUTMUX [SCOM]
CPFF is the Carry Propagate flip - flop.

DAD<3:0>
U<44:41>
XU<35:32>
G2<3:0>

Discrete Alteration of Data (See also SALU.)
Allows microprogram to alter operation of the data paths.

0	0	0	0	- NO-OP
.	.	.	1	- Allow odd adrs and DATOB for byte instr.
0	1	1	.	- check stack overflow
1	0	0	.	- generate CARRY IN to the ALU.
1	1	.	.	- ALU control f(IR) (can generate carry in to ALU) CMP or INC or ADC ∧ PS(C) or ROT(L) ∧ PS(C)
1	.	1	.	- inhibit DATO or DATOB and CLOCKOFF for (BITvCMPvTST)

Above bit patterns occur in combinations with other DAD bits.

DEST<1:0>
XU<63:62>
G3<15:14>

Destination (See also MSC, PPE and sections A.2 and A.3.)
The stack is loaded at the end of a cycle.
0 0 - OFF
0 1 - STACK ← DMUX
1 0 - RD ← Memory[Stack]
1 1 - Memory[Stack] ← DMUX

EMIT field
XU<79:64>
G4<15:0>

The EMIT field is a 16 bit field that can either be used as a constant or to implement some of the extended features of the 11/40E.
The use of the field is determined by the DEST and MSC codes.
When used as a constant the other functions are disabled.
(See sections A.2 and A.3. for EMIT field usage.)
(See also SCOM, PPE, CP, SC, RML, and LML.)

LML<3:0>
XU<67:64>
G4<3:0>

Left Mask Limit (See also RML.)
Masks data for RD BUS or EUBC BUS.
LML and RML are anded with the SMUX output.

				15 0
0	0	0	0	0000000000000001
0	0	0	1	0000000000000011
0	0	1	0	0000000000000111
		.		. .
		.		. .
1	1	1	0	0111111111111111
1	1	1	1	1111111111111111

MSC<2:0>
XU<61:59>
G3<13:11>

Mask-Shift Control (See also DEST, PPE and sections A.2 and A.3.)
Controls the state of the extension.

PPE
XU<77>
G4<13>

Push Pop Enable Allows stack operation.
(See also DEST, MSC, SC and section A.3.)
0 - PUSH/POP disabled
1 - PUSH/POP enabled

RIF<3:0>
U<12:9>
XU<3:0>
G0<3:0>

Register Immediate Field (See also SRI.)
Addresses general registers when enabled by SRI.
Console addresses for these registers are <777700:777717>.

Table of standard PDP-11 register designations:

0	0	0	0	- R0
0	0	0	1	- R1
0	0	1	0	- R2
0	0	1	1	- R3
0	1	0	0	- R4
0	1	0	1	- R5
0	1	1	0	- R[SP]
0	1	1	1	- R[PC]
1	0	0	0	- R[TEMP]
1	0	0	1	- R[SOURCE]
1	0	1	0	- R[DEST]
1	0	1	1	- R[IR]
1	1	0	0	- R[VECT]
1	1	0	1	- R[TEMPC]
1	1	1	0	- R[SP USER]
1	1	1	1	- R[ADRSC]

RML<3:0>
XU<71:68>
G4<7:4>

Right Mask Limit (See also LML.)
Masks data for RD BUS or EUBC BUS.
A 0 in the mask will mask out that bit.

				15 0
0	0	0	0	1000000000000000
0	0	0	1	1100000000000000
0	0	1	0	1110000000000000
.				. .
.				. .
1	1	1	0	1111111111111110
1	1	1	1	1111111111111111

NOTE: For information to pass through the masker, the mask bit must be one in the masks created by both LML and RML.

SALUM
U<37>
XU<28>
G1<12>

Selects ALU Mode of operation. (See also SALU.)
0 - Arithmetic mode
1 - Logical mode

SALU<3:0>
U<36:33>
XU<27:24>
G1<11:8>

Selects ALU function (See also SALUM and DAD.)

Arithmetic mode (SALUM = 0)

3	2	1	0	CARRY IN = 0	CARRY IN = 1
0	0	0	0	F←A	F←A + 1
0	0	0	1	F←A∨B	F←(A∨B) + 1
0	0	1	0	F←A∨¬B	F←(A∨¬B) + 1
0	0	1	1	F←- 1(2's comp)	F←0
0	1	0	0	F←A + A∧¬B	F←A + A∧¬B + 1
0	1	0	1	F←(A∨B) + A∧¬B	F←(A∨B) + A∧¬B + 1
0	1	1	0	F←A - B - 1	F←A - B
0	1	1	1	F←A∧¬B - 1	F←A∧¬B
1	0	0	0	F←A + A∧B	F←A + A∧B + 1
1	0	0	1	F←A + B	F←A + B + 1
1	0	1	0	F←(A∨¬B) + A∧B	F←(A∨¬B) + A∧B + 1
1	0	1	1	F←A∧B - 1	F←A∧B
1	1	0	0	F←A + A	F←A + A + 1
1	1	0	1	F←(A∨B) + A	F←(A∨B) + A + 1
1	1	1	0	F←(A∨¬B) + A	F←(A∨¬B) + A + 1
1	1	1	1	F←A - 1	F←A

SALU<3:0> continued

Logical mode (SALUM = 1)

3	2	1	0	
0	0	0	0	F←¬A
0	0	0	1	F←¬(A∨B)
0	0	1	0	F←¬A∧B
0	0	1	1	F←0
0	1	0	0	F←¬(A∧B)
0	1	0	1	F←¬B
0	1	1	0	F←A⊕B
0	1	1	1	F←A¬B
1	0	0	0	F←¬A∨B
1	0	0	1	F←¬(A⊕B)
1	0	1	0	F←B
1	0	1	1	F←A∧B
1	1	0	0	F←all 1's
1	1	0	1	F←A∨¬B
1	1	1	0	F←A∨B
1	1	1	1	F←A

SBAM
U<22>
XU<13>
G0<13>

Selects input to BUS ADDRESS MUX.
0 - BAMUX ← ALU
1 - BAMUX ← RD BUS

SBC<3:0>
U<32:29>
XU<23:20>
G1<7:4>

Allows selection of constants to the ALU through the BMUX.

OCTAL	NAME	VALUE	USE
00	TRAPS	0 000 000 000 0TT T00	T =f(STPM<4:2>.
01	CONST 1	0 000 000 000 000 001	General use.
02	CONST 2	0 000 000 000 000 010	General use.
03	CONST (1v2)	0 000 000 000 000 001	if word instr., sets CIN00
04	not used	0 000 000 000 000 000	
05	not used	0 000 000 000 000 000	
06	not used	0 000 000 000 000 000	
07	CONS INC	0 000 000 000 000 00C	C = EXAM(1) v DEP(1)
10	SR ADRS	1 111 111 101 111 000	also displayed on a cons time out.
11	PWR UP	7 777 077 076 543 200	prog address selected by jumpers W<7:2>.
12	CC MASK	0 000 000 000 001 111	Mask for condition code instrs.
13	SOB MASK	0 000 000 000 111 111	Mask in SOB instr.
14	SINCLK	0 000 000 000 0S0 000	S=0 for SWITCHCOUNT if SINCLK.
15	MM vector	0 000 000 010 101 000	KT violation trap vector.
16	MM const	0 000 000 000 000 0M0	M=1 if SM0 in MFP instr.
17	STACK 04	0 000 000 000 000 100	Preemptive stack pointer.

SBMH<1:0>
U<28:27>
XU<19:18>
G1<3:2>

Selects input to BMUX<15:8>. (See also SBML.)

0 0 - BMUX<15:8> ← B REGISTER<15:8>
0 1 - BMUX<15:8> ← B REGISTER<7>
1 0 - BMUX<15:8> ← B REGISTER<7:0>
1 1 - BMUX<15:8> ← CONSTANTS REGISTER<15:8>

SBML<1:0>
U<26:25>
XU<17:16>
G1<1:0>

Selects input to BMUX<7:0>. (See also SBMH)

0 0 - BMUX<7:0> ← B REGISTER<7:0>
0 1 - BMUX<7:0> ← B REGISTER<7:0>
1 0 - BMUX<7:0> ← B REGISTER<15:8>
1 1 - BMUX<7:0> ← CONSTANTS REGISTER<7:0>

SC<3:0>

XU<75:72>
G4<11:8>

The Shift Count field is either the rotate count to the right or it is a value to be loaded into the stack pointer.
Word rotation works as follows:

0 0 0 0 - P ONM LKJ IHG FED CBA
0 0 0 1 - A PON MLK JIH GFE DCB
0 0 1 0 - B APO NML KJI HGF EDC
0 0 1 1 - C BAP ONM LKJ IHG FED
. . . .
. . . .
1 1 1 1 - O NML KJI HGF EDC BAP

SCOM<1:0>

XU<79:78>
G4<15:14>

Selects COUT MUX (See also CP)
0 0 - ALU CARRY OUT <15>
0 1 - ALU CARRY OUT <7>
1 0 - PS (C) - Bit 0 of processor status word
1 1 - ALU <15>

SDM<1:0>
U<24:23>
XU<15:14>
G0<15:14>

Selects DMUX

0 0 - DMUX ← RD BUS
0 1 - DMUX ← UNIBUS DATA
1 0 - DMUX ← D REGISTER
1 1 - DMUX ← D REGISTER shifted right
with DMUX<15> ← D<C> (carry).

SPS<2:0>
U<40:38>
XU<31:29>
G1<15:13>

Selects loading and clocking the PS word.
Timing: P1 v P3

0	0	0	- DMUX gated to PS input allowing program loading.
0	0	1	- CLK (PS) C
0	1	0	- CLK PS(N,Z,V)
0	1	1	- CLK PS(N,Z,V,C)
1	0	0	- not used
1	0	1	- not used
1	1	0	- PS gated to BUS RD
1	1	1	- Load PS from DMUX

NOTE: (SPS=3) ANDed with CC instr also gates PS to BUS RD.
CC = Condition Code and is 0 000 000 010 1-- --- in the instruction register.

SRBA
U<14>
XU<5>
G0<5>

Allows BA<3:0> to be used as a source of general register address. (See note SRX.)
0 - NO-OP
1 - GENERAL REGISTER ADDRESS<3:0> ← BA<3:0>

SRD
U<15>
XU<6>
G0<6>

Allows IR<2:0> to be used as a source of general register address. (See note SRX.)
0 - NO-OP
1 - GENERAL REGISTER ADDRESS<2:0> ← IR<2:0>

SRI
U<13>
XU<4>
G0<4>

Allows RIF<3:0> to be used as a source of general register address. (See note SRX and RIF.)
0 - NO-OP
1 - GENERAL REGISTER ADDRESS<3:0> ← RIF<3:0>

SRS
U<16>
XU<7>
G0<7>

Allows IR<8:6> to be used as a source of general register address. (See note SRX.)
0 - NO-OP
1 - GENERAL REGISTER ADDRESS<2:0> ← IR<8:6>

SRX
U<16:13>
XU<7:4>
G0<7:4>

Mnemonic for the concatenation of SRBA, SRD, SRI and SRS.

UBF<4:0>
U<21:17>
XU<12:8>
G0<12:8>

Micro Branch Field. Allows microbranch condition to be tested (BUT). A successful test ORs BUBC bits into UPP<5:0>.

BUT	MNEMONIC CONDITION	PRINT	BUBC BITS
00	NO-OP		
01	CBR1 ¬HALT SWITCH	K5-7	--- --1

UBF<4:0> continued

BUT	MNEMONIC CONDITION	PRINT	BUBC BITS
02	CBR2 ¬HALT SWITCH	K5-7	--- --1
03	REG DEP REGISTER ADDRESS	K1-7	--- --1
04	REG EXAM ¬REGISTER ADDRESS	K1-7	--- --1
05	BEGIN ¬BEGIN	K5-6	--- --1
06	SWITCH SWITCH	K5-6	--- --1
07	INTR B INTR	K4-4	--- --1
10	HALT HALT SWITCH	K3-7	--- --1
11	MM FAULT ¬MM FAULT	KT-3	--- --1
12	D=0 D<15:00>=0	K1-7	--- --1
13	not used		
14	not used		
15	JSR ∨ JMP JSR	K3-5	--- --1
16	SERVICEC ∨ FETCHC SERVICE	K3-7	--- --1
17	IR03 IR03	K3-3	--- --1
20	BYTE∨SERVICE∨FETCH	K3-7	
	A) BYTE		--- -11
	B) SERVICE		--- -1-
	C) FETCH		--- --1
21	IR03,(BYTE ∧ SOURCE)	K3-7	
	A) SM0 ∧ ¬BYTE ∧ IR03		--- --1
	B) BYTE ∧ ¬SM0 ∧ IR03		--- -1-
	C) BYTE ∧ SM0 ∧ IR03		--- -11
22	BYTE ∧ SOURCE	K3-7	
	A) SM0 ∧ ¬BYTE		--- --1
	B) BYTE ∧ ¬SM0		--- -1-
	C) BYTE ∧ SM0		--- -11
23	not used		
24	CBR ∨ HALT HALT SWITCH	K3-7	--- -1-
25	BR,WAIT ∨ FETCH	K3-7	
	A) WAIT ∧ ¬BR		--- -11
	B) WAIT ∧ BR		--- -1-

UBF<4:0> continued

BUT	MNEMONIC CONDITION	PRINT	BUBC BITS
26	REQUESTS	K3-7	
	A) HALT SWT ∧ ¬CONSL ∧ ¬ERROR		--- --1
	B) (BR ∨ WAIT) ∧ ¬(HALT SWT ∧ ¬CONSL) ∧ ¬ERROR		--- -1-
	C) ¬(BR ∨ WAIT) ∧ ¬(HALT SWT ∧ ¬CONSL) ∧ ¬ERROR		--- -11
	NOTE: ERROR = PERR ∨ MM FAULT ∨ BERR ∨ PS(T) ∧ ¬RTT ∨ OVFLW ∨ PWRON		
27	SERVICE B ∨ FETCH OVLAP ∨ FETCH B	K3-7	
	A) OVLAP ∧ ¬SERVICE		--- --1
	B) SERVICE		--- -1-
30	SWITCHES	K5-6	
	A) START ∧ ¬(DEP ∨ CONT ∨ EXAM ∨ LOAD ADRS)		--- -1-
	B) DEP ∧ ¬(START ∨ CONT ∨ EXAM ∨ LOAD ADRS)		--- 1--
	C) EXAM ∧ ¬(START ∨ DEP ∨ CONT ∨ LOAD ADRS)		--- 1-1
	D) CONT ∧ ¬(START ∨ DEP ∨ EXAM ∨ LOAD ADRS)		--- 11-
	E) LOAD ADRS ∧ ¬(START ∨ DEP ∨ EXAM ∨ CONT)		--- 111
31	NOWR ∨ BYTEWR ∨ WORDWR		
	A) BYTE ∧ ¬(BIT ∨ CMP ∨ TST)	K3-7	--- --1
	B) (BIT ∨ CMP ∨ TST) ∧ ¬BYTE	K5-8	--- -1-
32	not used		

UBF<4:0> continued

BUT	MNEMONIC CONDITION	PRINT	BUBC BITS
33	OB ∨ INSTR 4	K3-7	
	A) ODD BYTE		--1 111
	B) NEG ∧ ¬ODD BYTE	--- --1	
	C) All single operand odd byte instr except NEG and SWAB.		--- -1-
	D) NEG odd byte		--- -11
	E) All double operands (except MOV and SUB) with SM0 and NOT odd byte.		--- 1--
	F) All double operands (except MOV and SUB) with SM0 and NOT odd byte.		--- 1-1
	G) SUB ∧ SM0 ∧ ¬DM0 ∧ ¬ ODD BYTE		--- 11-
	H) SUB ∧ ¬SM0 ∧ ¬DM0		--- 111
	I) All odd byte double operands (except MOV and SUB) with SM0.		--1 ---
	J) All odd byte double operands (except MOV and SUB) with ¬SM0.		--1 --1
	K) (ROR/ROL/ASR/ASL) ∧ ¬DM0		--1 -1-
	L) ROR(B)/ROL(B)/ASR(B)/ASL(B) ∧ ¬DM0		--1 -11
	M) SXT ∧ ¬DM0		--1 1--
	N) SWAB ∧ ¬DM0		--1 11-
34	INSTR 4		
	BUT 34 is identical to BUT 33 without regard to the odd byte condition.		
35	OB ∨ INSTR 3	K3-7	
	A) ODD BYTE		--1 11-
	B) Identical to *'ed items in BUT 37 ∧ ¬ODD BYTE		
	C) SUB ∧ ¬SM0 ∧ DM0 ∧ ¬ODD BYTE		--- --1
36	INSTR 3		
	BUT 36 is identical to BUT 35 without regard to the ODD BYTE condition.		

UBF<4:0> continued

BUT	MNEMONIC CONDITION	PRINT	BUBC BITS
37	INSTR 1		
	HALT		-1- -1-
	WAIT		--1 1--
	RTI		--- --1
	BPT		-1- 11-
	IOT		-1- 11-
	RESET		-1- 111
	RTT		--- --1
	JSR/JMP (Rn)		1-1 --1
	JSR/JMP (Rn)+		1-1 -1-
	JSR/JMP @(Rn)+		1-1 -11
	JSR/JMP -(Rn)		1-1 1--
	JSR/JMP @-(Rn)		1-1 1-1
	JSR/JMP x(Rn)		1-1 11-
	JSR/JMP @x(Rn)		1-1 111
	RTS		-1- 1--
	NOP		--1 11-
	CCC		--1 11-
	SEC		--1 111
	SWAB RN		-11 1--
	BRANCH INSTR. (cond. not met)		--1 ---
	BRANCH INSTR. (cond. met)		--1 --1
	* The following applies also to BUT 35 B until a second * appears.		
	All single operand, SWAB ∧ ¬Rn, ROTATEs, SHIFTs, all double ops. (except MOVE) with SOURCE MODE 0.		
	xx xx (Rn)		11- --1
	xx xx (Rn)+		11- -1-
	xx xx @(Rn)+		11- -11
	xx xx -(Rn)		11- 1--
	xx xx @-(Rn)		11- 1-1
	xx xx x(Rn)		11- 11-
	xx xx @x(Rn)		11- 111

UBF<4:0> continued

MNEMONIC	BUBC BITS
BUT 37 continued	
MOV Rn, Rn	111 ---
MOV Rn, (Rn)	111 --1
MOV Rn, (Rn)+	111 -1-
MOV Rn, @(Rn)+	111 -11
MOV Rn, -(Rn)	111 1--
MOV Rn, @-(Rn)	111 1-1
MOV Rn, x(Rn)	111 11-
MOV Rn, @x(Rn)	111 111
* Here ends the section relating to BUT 35.	
ILLEGAL INSTR	1-1 --
MFPI, MTPI, MFPD, MTPD	-11 11-
SOB	-11 ---
SXT Rn	-11 -1-
SUB Rn, Rn	--- -11
NEG Rn	--- 1-1
All double operand (except MOV and SUB) with Dest mode and Source mode = 0:	
xx 0R 0R	--- -1-
All single operand (except MOV and SUB) with Dest mode 0:	
xx xx 0R	--- 1--
All double operand with Source mode not 0:	
xx (Rn) xx	1-- --1
xx (Rn)+ xx	1-- -1-
xx @(Rn)+ xx	1-- -11
xx -(Rn) xx	1-- 1--
xx @-(Rn) xx	1-- 1-1
xx x(Rn) xx	1-- 11-
aa @x(Rn) xx	1-- 111
ROR/ROL/ASR/ASL Rn	--- 11-
(ROR(B)/ROL(B)/ ASR(B)/ASL(B)) Rn	--- 111

UPF<7:0> The 8 bit next address field.
U<7:0> Used to specify address of next microinstruction to be
XU<55:48> executed but may be modified as a result of a branch test.
G3<7:0> (See also UBF and XUPF.)
NOTE: The UPF bits are complemented while XUPF bits are not.
UBC is MicroBranch Control.
The UPF field is ORed with BUBC from the BUT MUX (K3-2) in the processor and the EUBC (K2-2,3) lines from the extension. This allows conditional microbranches.

WRH Allows writing DMUX<15:8> into the general registers.
U<52> 0 - NO-OP
XU<43> 1 - GENERAL REGISTER<15:8> (as selected by SRX) ← DMUX<15:8>
G2<11> Timing: P1 v P3

WRL Allows writing DMUX<7:0> into the general registers.
U<51> 0 - NO-OP
XU<42> 1 - GENERAL REGISTER<7:0> (as selected by SRX) ← DMUX<7:0>
G2<10 Timing: P1 v P3

XUPF<10:8> Extended UPF field. (See also UPF.)
XU<58:56> XUPF<9:8> are concatenated with EUPP<7:0> for the RAM address.
G3<10:8>

0 0 0 - Enables Bootstrap PROM on first entering the extension (an automatic hardware feature).
If already in the extension this combination will cause a branch back to the 11/40 ROM.
NOTE: The last microinstruction before returning to the 11/40 ROM cannot use the extended features.
(an AUTOMATIC hardware clear)

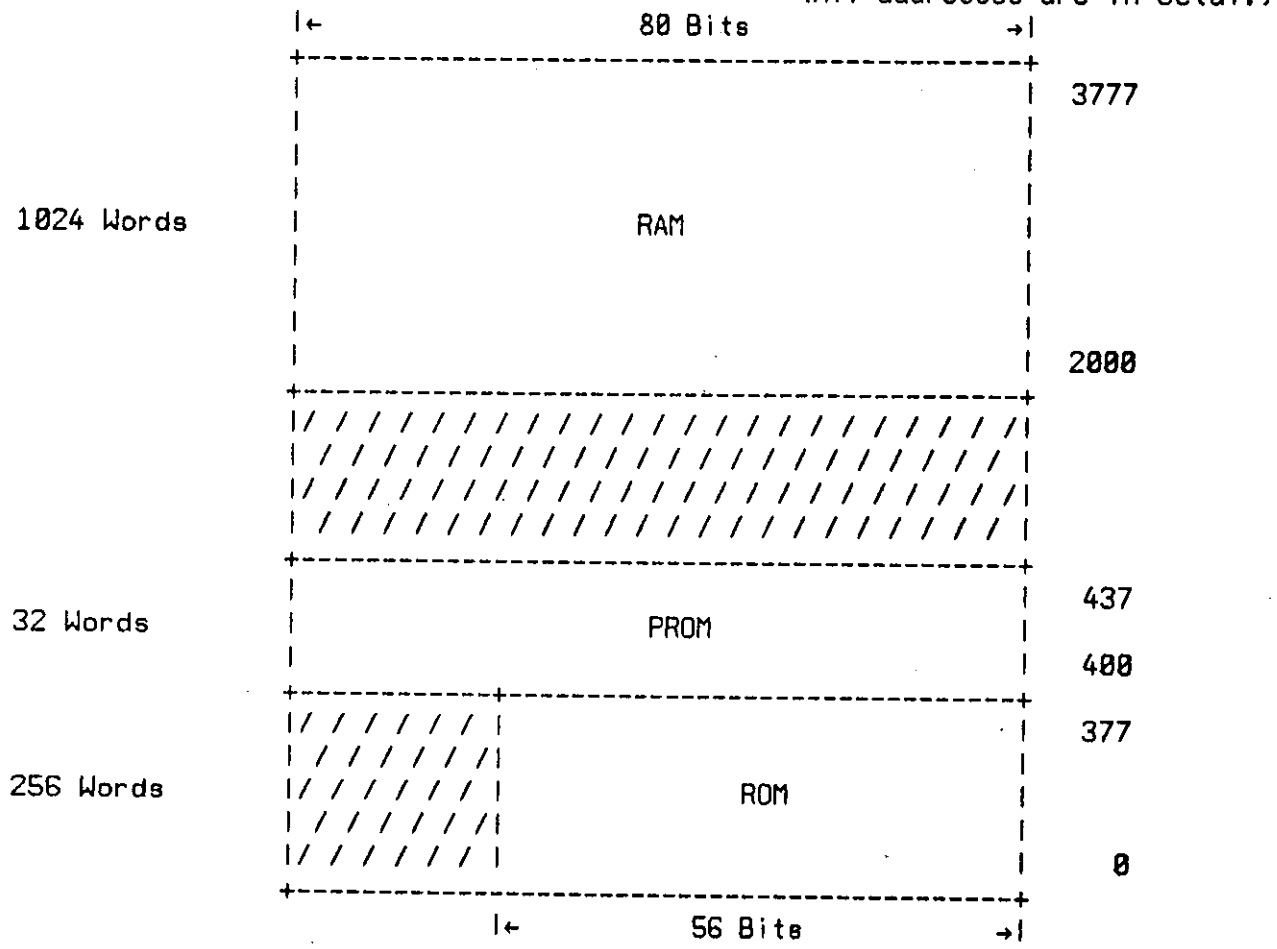
0 0 1 - Enables bootstrap PROM. During a RAMread or RAMwrite the PROM is only enabled after the P2 pulse of a P3 cycle.
0 1 0 - illegal
0 1 1 - illegal
1 . . - Enables reading of the 80 bit RAM.
1 0 0 - XUPF<9:8> concatenated with EUPP<7:0>
1 0 1 to address the 1K by 80 bit
1 1 0 microwords. (See section A.2.)
1 1 1

A.2 Address Space

Addressing U Memory

Using XUPF<10:8>,UPF<7:0> as address of next 80 bit U word to execute.

(All addresses are in Octal.)



Using the contents of the stack to address a 16 bit word in the RAM.

Group	4	3	2	1	0	U Word Stack<13:3>
	77770	37776	37774	37772	37770	3777
	77760	37766	37764	37762	37760	3776
						.
						.
						.
	60020	20026	20024	20022	20020	2002
	60010	20016	20014	20012	20010	2001
	60000	20006	20004	20002	20000	2000
	57770	17776	17774	17772	17770	1777
If 2K	57760	17766	17764	17762	17760	1776
Chips						.
Exist						.
	40000	6	4	2	0	0

The group number is determined by STACK<14,2:1>.

0 0 0 - group 0
0 0 1 - group 1
0 1 0 - group 2
0 1 1 - group 3
1 0 0 - group 4
1 0 1 - illegal
1 1 0 - illegal
1 1 1 - illegal

A.3 DEST/MSC Functions

Interaction of Dest (Destination) and MSC (Mask-Shift Control) fields. NOTE: Actions occur from left to right. The stack is loaded at the end of a cycle. Some timings are indicated.

DEST	MSC	Action
00	000	OFF
00	001	RD ← EMIT
00	010	EUBC ← EMIT
00	011	Push Stack; STACK ← EMIT (P2 or P3)
00	100	STACK ← EMIT (P1 or P2 or P3)
00	101	RD ← Stack Pointer
00	110	RD ← Stack; Pop (if PPE=1)
00	111	EUBC ← Stack; Pop (if PPE=1)
01	000	Push (if PPE=1); Stack ← DMUX (if PPE=1 P2 or P3 only)
01	001	Push; RD ← EMIT; Stack ← DMUX (P2 or P3)
01	010	Push; EUBC ← EMIT; Stack ← DMUX (P2 or P3)
01	011	Push; (P1 or P2 or P3)
01	100	Pop
01	101	Push (if PPE=1); RD ← Stack Pointer; Stack ← DMUX (if PPE = 1 P2 or P3 only)
01	110	RD ← Stack; Stack ← DMUX
01	111	STACK POINTER ← SC<3:0>
10	000	OFF
10	001	OFF
10	010	OFF
10	011	OFF
10	100	RD ← Memory[Stack]; Pop (if PPE=1)
10	101	OFF
10	110	OFF
10	111	OFF
11	000	Memory[Stack] ← DMUX; Pop (if PPE=1)
11	001	RD ← EMIT; Memory[Stack] ← DMUX; Pop
11	010	EUBC ← EMIT; Memory[Stack] ← DMUX; Pop
11	011	OFF
11	100	OFF
11	101	RD ← Stack Pointer; Memory[Stack] ← DMUX; Pop (if PPE=1)
11	110	RD ← Stack; Memory[Stack] ← DMUX; Pop (if PPE=1)
11	111	EUBC ← Stack; Memory[Stack] ← DMUX; Pop (if PPE=1)

Push: (pushes with writes to stack can be P2 or P3 only)
Stack Pointer ← Stack Pointer-1 (push) then
Action

Pop:
Action then
Stack Pointer ← Stack Pointer+1 (pop)

Appendix B. Standard ROM in MICRO/40

```
!  This is a transcription of the standard PDP-11 emulator
!  in MICRO/40.  The sequence and labels of the micro-
!  instructions closely follow the flowcharts in the KD11-A
!  Processor Engineering Drawings.
!  This microcode is available on the CMU-10A as
!  ROM256.MIC[N200MU00] and can be used as a starting
!  point when constructing a modified PDP-11 instruction
!  set on the PDP-11/40E microprocessor.
!
!  Original Version:  Summer 1975  Rajan Modi
!  Revision 1:        Autumn 1975  Guy Almes

require defs.mic[n200mu00]

f/ir/         := dad=14$          ! alu function of ir
allow.odd     := dad=1$           ! allow an odd address
check.ovflo   := dad=6$           ! check stack overflow
ovflo/odd     := dad=7$
dato/datob    := dad=17; dato$
inhibit       := dad=12$          ! inhibit dato/clkoff on TST or CMP
clkoff/ovlap  := clk=0$           ! clkoff if overlap fetch
awbby         := bus=2$           ! await bus busy

disp.rir    := sdm=0; srx=1; rif=13$     ! set data to be displayed
disp.rpc    := sdm=0; srx=1; rif=7$
disp.vect   := sdm=0; srx=1; rif=14$
disp.adrsc  := sdm=0; srx=1; rif=17$
disp.rdf    := sdm=0; srx=4$
disp.d      := sdm=2$
disp.bus    := sdm=1$

begin   goto FET02

set     ! but 25 @ SER07
   start
      SER09: p1; disp.rir; clkoff
      SER10: p3; b,vect←unibus; but 7
      SER11: p1; disp.vect; goto SER04
   end
      SER05: p1; disp.rir; goto TRP03
      FET02: p1; ba←rpc; dati; clkoff; goto FET03
      SER02: p3; disp.rir; but 26; goto SER05
tes

set     ! but 20 @ MOV19
   FET01: p1; ba←rpc; dati; clkoff; goto FET03
   FET03: p1; b,ir,rir←unibus; goto FET04
```

```
   SER01: p3; disp.rir; but 26; goto SER05
   MOV21: p2; d←b.el; but 27; disp.d; goto MOV20
tes

FET04: p2; ba,d←rpc+2; but 37; dati ! if ovlap fetch
FET05: p1; rpc←d
set      ! but 37 @ FET04
      FET07: p1; disp.rir; goto TRP06
   start
      RTI00: p3; ba←rsp; d←rsp+2; dati; rsp←d; clkoff
      RTI01: p1; rpc←unibus
      RTI02: p3; ba←rsp; d←rsp+2; dati; rsp←d; clkoff
      RTI03: p1; ps←unibus; goto SER02
   end
   start
      DOP02: p1; b←rdf; but 31
      DOP03: p2; d←rsf; f/ir/; but 27
         set    ! but 31 @ DOP02
            DOP19: p1; rdf←d; n.z.v.c.; clkoff/ovlap; goto FET01
            DOP18: p1; r[df].l←d; n.z.v.c.; clkoff/ovlap; goto FET01
            DOP17: p1; n.z.v.c.; disp.d; clkoff/ovlap; goto FET01
         tes
   end
   start
      DOP00: p1; b←rsf
      DOP01: p2; d←rdf-b; but 27; goto DOP19
   end
   start
      SSL02: p2; d←rdf; f/ir/; but 31
      SSL03: p1; c.; disp.d; but 27; goto DOP19
   end
   start
      SSL00: p1; b←rdf; but 31
      SSL01: p2; d←0-b; disp.d; but 27; goto DOP19
   end
   start
      RSR00: p3; d←rdf; c.; b←dshift
      RSR01: p3; d←b; rdf←d; but 27
      RSR05: p1; n.z.v.; clkoff/ovlap; disp.rdf; goto FET01
   end
   start
      RSR02: p1; b←rdf
      RSR03: p3; d←b.lz; c.; b←dshift
      RSR04: p2; d←b.hh; r[df].l←d; but 27; goto RSR05
   end
   start
      NBR00: p1; disp.rir; but 16
      SOB06: p1; disp.rir; goto FET02
   end
   start
```

```
      BRA00: p2; d←rpc+b.el
      BRA01: p1; rpc←d; but 16
      BRA02: p3; d←rpc+b.el; rpc←d; goto FET02
   end
   start
      MRK00: p3; d←ir+b; b←d
      MRK01: p3; d,ba←rpc+b.el; dati; rpc←d
      MRK02: p2; d←rpc+2
      MRK03: p1; rsp←d; clkoff
      MRK04: p3; d←r5; r5←unibus; but 16
      MRK05: p1; rpc←d; goto FET02
   end
noop  ! hole
      SER00: p3; disp.ir; but 26; goto SER05
noop  ! hole
   start
      CCC00: p3; d←ir and 17; b←d
      CCC01: p3; d←not b; b←d; but 16
      CCC02: p3; d←ps and b; n.z.v.c.; ps←d; goto FET02
   end
   start
      SCC00: p3; d←ir and 17; b←d; but 16
      SCC01: p3; d←ps or b; disp.d; goto FET02
   end
   start
      DOP15: p1; b←rdf; but 31
      DOP16: p2; d←rsrc; f/ir/; but 27; goto DOP19
   end
   start
      DOP13: p1; b←rsrc
      DOP14: p2; d←rdf-b; but 27; goto DOP19
   end
      CON00: p3; ba←rpc; disp.d; goto CON12
      SER03: p3; disp.ir; but 26; goto SER05
   start
      RTS00: p2; d←rdf
      RTS01: p1; rpc←d
      RTS02: p3; ba←rsp; d←rsp+2; dati; rsp←d; clkoff; but 16
      RTS03: p1; rdf←unibus; goto FET02
   end
noop  ! hole
      TRP06: p3; d←c[0]; vect←d; goto TRP08
   start
      RST00: p1; d←r0; disp.d
      RST01: p3; disp.d; clkoff; but 2
      RST02: p1; disp.d
         set    ! but  2 @ RST01
            RST03: p1; disp.d; goto CON01
            RST04: p1; disp.d; goto FET02
         tes
```

```
    end
    start
       SOB00: p2; d←rsf-1
       SOB01: p1; rsf←d; but 12
       SOB02: p2; d←ir and 77; b←d
          set    ! but 12 @ SOB01
             start
                SOB03: p3; d←rpc-b; rpc←d; but 16
                SOB04: p3; d←rpc-b; rpc←d; goto FET02
             end
                SOB05: p1; disp.d; but 16; goto SOB06
          tes
    end
noop  ! hole
       SXT00: p2; d←-1; f/ir/; disp.d; but 27; goto DOP19
noop  ! hole
    start
       SWB00: p1; b←rdf
       SWB01: p2; d←b.lh; disp.d; but 27; goto DOP19
    end
noop  ! hole
KT: noop !?
    start
       SRC16: p2; d←b.hh; disp.rir; but 36
       SRC17: p1; b,rsrc←d; goto DOP15
    end
noop  ! hole
    start
       SRC00: p1; ba←rsf; dati; allow.odd
       SRC14: p1; clkoff; disp.rir; but 35
       SRC15: p1; b,rsrc←unibus; goto DOP15
    end
    start
       SRC01: p2; ba←rsf; dati; allow.odd; d←rsf+c[3]
       SRC03: p1; rsf←d; clkoff; but 35; goto SRC15
    end
    start
       SRC04: p3; ba←rsf; dati; d←rsf+2; rsf←d; clkoff
       SRC12: p1; b,rsrc←unibus
       SRC13: p1; ba←rsrc; dati; allow.odd; goto SRC14
    end
       SRC02: p2; d,ba←rsf-c[3]; dati; allow.odd; goto SRC14
       SRC05: p3; d,ba←rsf-2; dati; rsf←d; clkoff; goto SRC12
    start
       SRC06: p3; d←rpc+2; rpc←d; clkoff
       SRC07: p1; b,rsrc←unibus
       SRC08: p2; ba←rsf+b; dati; allow.odd; goto SRC14
    end
    start
       SRC09: p3; d←rpc+2; rpc←d; clkoff
```

```
      SRC10: p1; b,rsrc←unibus
      SRC11: p2; ba←rsf+b; dati; clkoff; goto SRC12
   end
      TRP07: p3; d←c[0]; vect←d; goto TRP08
   start
      JMP00: p2; d←rdf; but 15
      JMP04: p1; b,temp←d
         set    ! but 15 @ JMP00
            start
               JMP12: p2; d←temp; but 16
               JMP13: p1; rpc←d; goto FET02
            end
            start
               JSR00: p3; d,ba←rsp-1-1; rsp←d; check.ovflo
               JSR01: p2; d←rsf; dato; clkoff
               JSR02: p2; d←rpc
               JSR03: p1; rsf←d; goto JMP12
            end
         tes
   end
   start
      JMP01: p3; d←rdf+2; rdf←d
      JMP02: p2; d←rdf-2; but 15; goto JMP04
   end
   start
      JMP05: p3; ba←rdf; d←rdf+2; dati; rdf←d; clkoff; but 15
      JMP11: p1; b,temp←unibus; goto JMP12
   end
      JMP03: p3; d←rdf-2; rdf←d; but 15; goto JMP04
      JMP06: p3; d,ba←rdf-2; dati; rdf←d; clkoff; but 15;-
                                                  goto JMP11
   start
      JMP08: p3; d←rpc+2; rpc←d; clkoff
      JMP14: p1; b,temp←unibus
      JMP15: p2; d←rdf+b; but 15; goto JMP04
   end
   start
      JMP07: p3; d←rpc+2; rpc←d; clkoff
      JMP09: p1; b,temp←unibus
      JMP10: p2; d,ba←rdf+b; dati; clkoff; but 15; goto JMP11
   end
   start
      MOV19: p2; d←rsrc; but 20
      MOV20: p1; rdf,b←d; n.z.v.c.; clkoff/ovlap; goto FET01
   end
   start
      DST00: p1; ba←rdf; datip; ovflo/odd
      DST14: p1; disp.rir; clkoff; but 33
      DST15: p1; b,rdst←unibus
         set    ! but 33 @ DST14
```

```
            start
               SSL06: p2; d←rdst; f/ir/; dato/datob; disp.d
               SSL10: p1; c.; disp.d
               DOP12: p1; n.z.v.c.; clkoff; inhibit; disp.d;-
                                                     but 16
               DOP20: p1; disp.d; goto FET02
            end
               SSL04: p2; d←0-b; dad=11; dato; disp.d; goto DOP12
            start
               SSL08: p2; d←rdst; f/ir/
               SSL12: p1; c.; disp.d
               SSL11: p1; b←d; n.z.v.c.
               SSL09: p2; d←b.lz; datob; disp.d;-
                        dad=13; clkoff; but 16; goto DOP20
            end
               SSL07: p2; d←0-b; disp.d; goto SSL12
               DOP07: p2; d←rsf; f/ir/; dato/datob; goto DOP12
               DOP08: p2; d←rsrc; f/ir/; dato/datob; goto DOP12
            start
               DOP04: p1; b←rsf
               DOP06: p2; d←rdst-b; dato; goto DOP12
            end
               DOP05: p1; b←rsrc; goto DOP06
            start
               DOP09: p2; d←rsf; f/ir/
               DOP22: p1; b←d; c.
               DOP21: p1; b←d; n.z.v.c.
               DOP11: p2; d←b.lz; datob; disp.d; dad=13;-
                                 clkoff; but 16; goto DOP20
            end
               DOP10: p2; d←rsrc; f/ir/; goto DOP22
            start
               RSR06: p3; d←b; c.; b←dshift
               RSR07: p2; d←b; dato; disp.d
               RSR10: p1; n.z.v.; clkoff; disp.d; but 16
               RSR11: p1; disp.d; goto FET02
            end
            start
               RSR08: p2; d←b.lz; c.; b←dshift
               RSR09: p2; d←b.hh; datob; disp.d; goto RSR10
            end
               SXT01: p2; d←-1; f/ir/; dato; disp.d; goto DOP12
                      noop !hole
               SSL05: p2; d←b.lh; dato; disp.d; goto DOP12
            start
               DST16: p2; d←b.hh; disp.rir; but 34
               DST17: p1; b,rdst←d; goto SSL06
            end
         tes
end
```

```
start
   DST01: p2; ba←rdf; datip; ovflo/odd; d←rdf+c[3]
   DST03: p1; rdf←d; clkoff; but 33; goto DST15
end
start
   DST04: p3; ba←rdf; dati; d←rdf+2; rdf←d; clkoff
   DST12: p1; b,rdst←unibus
   DST13: p1; ba←rdst; datip; allow.odd; goto DST14
end
   DST02: p2; d,ba←rdf-c[3]; datip; ovflo/odd; goto DST03
   DST05: p3; d,ba←rdf-2; dati; rdf←d; clkoff
start
   DST07: p3; d←rpc+2; rpc←d; clkoff; but 17
   DST09: p1; b,rdst←unibus
      set    ! but 17 @ DST07
         DST10: p2; ba←rdf+b; datip; ovflo/odd; goto DST14
         DST11: p2; ba←rdf+b; dati; clkoff; goto DST12
      tes
end
   DST06: p3; ba←rpc; dati; d←rpc+2; rpc←d; clkoff;-
                                          but 17; goto DST09
   MOV18: p2; d←rsf; but 20; goto MOV20
start
   MOV00: p2; d,ba←rdf; ovflo/odd; but 22
   MOV07: p1; rdf←d
      set    ! but 22 @ MOV00
            MOV16: p2; d←rsrc; dato
            MOV17: p2; d←rsf; dato
         start
            MOV14: p1; b←rsrc
            MOV15: p3; d←b.lz; datob; rsrc←d
         end
            MOV13: p1; b←rsf; goto MOV15
      tes
   MOV22: p3; n.z.v.c.; disp.d; inhibit; clkoff; but 16;-
                                                 goto DOP20
end
   MOV01: p2; ba←rdf; d←rdf+c[3]; ovflo/odd; but 22;-
                                             goto MOV07
start
   MOV03: p3; ba←rdf; d←rdf+2; dati; rdf←d; clkoff
   MOV11: p1; b,rdst←unibus; but 22
   MOV12: p1; ba←rdst; allow.odd; goto MOV16
end
   MOV02: p2; d,ba←rdf-c[3]; ovflo/odd; but 22; goto MOV07
   MOV04: p3; d,ba←rdf-2; dati; rdf←d; clkoff; goto MOV11
start
   MOV06: p3; d←rpc+2; rpc←d; clkoff; but 17
   MOV08: p1; b,rdst←unibus; but 21
      set    ! but 17 @ MOV06
```

```
            MOV09: p2; ba←rdf+b; ovflo/odd; goto MOV16
            MOV10: p2; ba←rdf+b; dati; clkoff; allow.odd; goto MOV11
         tes
   end
      MOV05: p3; ba←rpc; dati; d←rpc+2; rpc←d; clkoff; but 17;-
                                                        goto MOV08
tes


set      ! but 26 @ SER02
   start
      TRP03: p3; d←c[0]; vect←d
      TRP02: p2; ba←vect+2; dati; clkoff; goto TRP09
   end
      CON01: p3; ba←rpc; disp.d; goto CON12
   start
      SER06: p1; disp.rir; awbby; clkoff
      SER07: p2; disp.rir; awbby; but 25
      SER08: p1; disp.rir; goto SER09
   end
      FET00: p1; ba←rpc; dati; clkoff; goto FET03
tes


set      ! but  7 @ SER10
      SER04: p3; disp.rir; but 26; goto SER05
   start
      TRP08: p2; ba←vect+2; dati; clkoff
      TRP09: p1; temp←unibus
      TRP10: p3; d,ba←rsp-b; check.ovflo; rsp←d
      TRP11: p2; d←ps; dato; clkoff
      TRP12: p1;
      TRP13: p3; d,ba←rsp-b; check.ovflo; rsp←d
      TRP14: p2; d←rpc; dato; clkoff
      TRP15: p1; ps←temp
      TRP16: p1; ps←temp; but 4
      TRP20: p1; ba←vect; dati; clkoff; but 1
      TRP21: p1; rpc←unibus; but 3; goto CON00
   end
tes


set      ! but 30 @ CON10
   start
      CON05: p3; disp.d
      CON13: p3; disp.d; but 6
      CON06: p1; disp.d; goto CON04
```

```
      CON08: p3; temc←d; but 12
      CON09: p3; d←temc+1; temc←d; goto CON08
      CON10: p3; disp.rpc; but 30
      CON11: p3; d←adrsc; goto CON05
   end
   start
      EXM06: p2; d←177570; dati; disp.d; clkoff
      EXM07: p1; b←unibus
      EXM08: p2; d←b; disp.d; goto CON05
   end
   start
      STA00: p1; rpc←d; but 10
      STA01: p1; disp.rpc; goto FET02
   end
      LAD03: p1; d←0; disp.d; goto CON05
   start
      DEP00: p1; ba←adrsc
      DEP01: p3; disp.d; but 3
      DEP02: p3; d,ba←adrsc+c[7]; adrsc←d
         set    ! but  3 @ DEP01
            DEP03: p3; d,ba←adrsc+c[7]; adrsc←d; goto DEP04
            DEP04: p2; ba←177570; dati; disp.adrsc; clkoff
         tes
      DEP05: p1; b←unibus
      DEP06: p1; ba←adrsc
      DEP07: p3; d←b; allow.odd; disp.d; but 3
      DEP08: p1; disp.d
         set    ! but  3 @ DEP07
            DEP09: p1; dato; disp.d; clkoff; goto CON05
            DEP10: p1; rba←d; goto CON05
         tes
   end
   start
      EXM00: p1; ba←adrsc
      EXM01: p3; disp.d; but 4
      EXM02: p3; d,ba←adrsc+c[7]; adrsc←d
         set    ! but  4 @ EXM01
            EXM04: p3; disp.adrsc; allow.odd; but 4
            EXM03: p3; d,ba←adrsc+c[7]; adrsc←d; goto EXM04
         tes
      EXM05: p2; d←rba; goto CON05
   end
      CNT00: p1; disp.rpc; goto SER02
   start
      LAD00: p2; ba←177570; dati; disp.bus; clkoff
      LAD01: p3; adrsc←unibus; but 5
      LAD02: p2; d,ba←adrsc; disp.d; goto STA00
   end
tes
```

```
CON12: p3; d←ps; awbby; but 24
CON02: p3; ps←d; awbby; clkoff
   set  ! but 24 @ CON12
      CON03: p2; d←r0; disp.d; goto CON04
             noop; goto 0 !hole
      CON04: p3; disp.d; but 6; goto CON06
      CON07: p2; d←b; sbm=17; sbc=14; disp.d; goto CON08
   tes
finis
```

Appendix C. The Bootstrap PROM

```
!       Proposed PROM Code -- 2 December 1974
!                             -- rev 1 May 1975
!                             -- rev 9 June 1975
!
!       The use of a single opcode, 000007, for selection of
!  PROM functions offers three advantages:
!
!       a)  It uses the only opcode with no address bits,
!       b)  It reduces the delay in getting to 2000 in normal cases,
! and   c)  It leaves more of the PROM open for utility goto's.
!
        require defs.mic
        lowlim=2400
 .=2400; tos←ps          ! push ps onto stack for addr space check
        case tos<7>      ! test bit 7 to detect whether user or Hydra
        d←7 xor b        ! d←ir xor 000007
    set
            tos←rir; goto 2000          ! execute user instruction
            skipzero                    ! in Hydra; detect load/store
    tes
        d←r2
    set
            tos←rir; goto 2001          ! execute Hydra instruction
            tos←r1; skipzero            ! load/store, since ir=000007
    tes
        d←not r0                        ! assume a RAM write
    set
        start                           ! r2 0  RAM read
            d←RAM[tos]; but 16
            r0←d;       goto 16
        end
        RAM[tos]←d;     goto exitt      ! r2=0  RAM write
    tes

rdram:  d←ram[s]; goto retsub           ! read  from the RAM
wrram:  ram[s]←d; goto retsub           ! write from the RAM
        eubc←tos; goto exxit            ! provide two kinds of return
retsub: eubc←s;   goto exxit
exxit:  goto 0

exitt:  but 16; eubc←16; goto exxit  ! provide an exxit to the ROM

 .=2402; goto 2002 ! provide some .=40x; goto's for RAM readers
 .=2403; goto 2003
 .=2404; goto 2004
 .=2405; goto 2005
 .=2406; goto 2006
```

```
.=2407; goto 2007
.=2410; goto 2010
.=2411; goto 2011
.=2412; goto 2012
.=2413; goto 2013
.=2414; goto 2014
.=2415; goto 2015
.=2416; goto 2016
.=2417; goto 2017
      finis
```

Appendix D. Vector Instruction Set

The following microcode implements a set of four vector instructions: 'Vector Move', 'Vector Compare', 'Broadcast', and 'Checksum'. Each is interruptable and has both a word and a byte varient. The instruction is two words long. The first word must have a free 'DF' field; the indicated register provides a count register, which is decremented for each item processed until it reaches zero. The second word is of the form 'bc0s0d', where 'b'=1 iff 'item'=byte; 'c' indicates the op-code; and 's' and 'd' specify the source and destination registers, respectively.

Vector Move replaces the code sequence:

```
loop:   MOV(B)    (S)+,(D)+
        DEC       K
        BNE       loop
```

Vector Compare replaces the code sequence:

```
loop:   DEC       K
        CMP(B)    (S)+,(D)+
        BNE       out
        TST       K
        BNE       loop
out:    ...
```

Broadcast replaces the code sequence:

```
loop:   MOV(B)    S,(D)+
        DEC       K
        BNE       loop
```

Checksum replaces the code sequence:

```
loop:   ADD       (S)+,D
        ADC       D
        DEC       K
        BNE       loop
```

This microcode is not naive; understanding it requires thorough attention to the details of BUT 35, byte handling, and flow of control constructs. Note that the first instruction to be executed is an invocation of macro 'Vector'; this macro would be invoked in the instruction decoding microcode of the RAM. Note also that the macro's of "DEFS.MIC[N200MU00]" and a macro 'Illegal', which disposes of illegal opcodes, are used.

```
! Interruptible Vector Instruction Set -- 28.IX.75
!
Vector :=       ba←rpc; dati; prop; goto Vec0$
nextsrc:=       ba←rsf; d←rsf+c[3]; rsf←d; dad=1; dati$
opcode :=       case tos<14:12>$
```

```
oddbyte:=       but 35; case tos<15>↑6$
decount:=       d←count-1; count←d$
writew :=       dato; clkoff$
writeb :=       datob; clkoff; dad=1$
nextdst:=       ba←rdf; d←rdf+c[3]; rdf←d; dad=1$
count  :=       r[11]$
cmpare :=       n.z.v.c.; skipzero; goto vec5$
cksum  :=       n.z.v.c.; prop; sbc=10; opcode; goto 0$

Vec0:   p3; d←rdf; skipzero; clkoff
        ir,tos←unibus

  set   ! initial zero count
        count←d; opcode
vec1:   d←rpc+2; rpc←d; goto vec6       ! complete the instruction
  tes

  set   ! catch interrupt requests
vec2:   noop                            ! proceed
        d←rpc-2; rpc←d; goto vec6       ! suspend the instruction
  tes

  set   ! opcode decoding

        Illegal                                 ! 00 - undefined
        Illegal                                 ! 01 - undefined

    start                                       ! 02 - Block Transfer
        nextsrc
        decount; skipzero; clkoff
        b←unibus; oddbyte
      set       ! check for zero count
vec3:   nextdst; but 16; case vec2      ! count   0
        nextdst; case vec1              ! count = 0
      tes

.=2067; d←b   ; writew; opcode; goto 0 ! word
.=2167; d←b.ll; writeb; opcode; goto 0 ! even byte
.=2177; d←b.hh; writeb; opcode; goto 0 ! odd byte
   end ! Block Transfer

   start                                        ! 03 - Compare
        nextsrc
        oddbyte; clkoff
        temp,b←unibus

.=2467; nextdst; dati; goto vec4        ! word
.=2567; d←b.cl; temp←d; goto 2467       ! even byte
.=2577; d←b.ch; temp←d; goto 2467       ! odd byte
```

```
vec4:   decount; clkoff; oddbyte
        b←unibus; skipzero

 .=2667; d←temp-b   ; cmpare            ! word
 .=2767; d←temp-b.cl; cmpare            ! even byte
 .=2777; d←temp-b.ch; cmpare            ! odd byte

      set       ! check exhausted count
vec5:   but 16                  ! count   0
        goto vec1               ! exit: count = 0
      tes
      set       ! check for disagreement on compare
        goto vec1               ! exit: mismatch
        goto vec2; opcode       ! continue
      tes
    end ! Compare

    start                                       ! 04 - Broadcast
        decount; skipzero
        b←rsf; case tos<15>↑6; goto vec3
    end ! Broadcast

    start                                       ! 05 - Checksum
        nextsrc
        decount; skipzero; clkoff
        b←unibus; oddbyte
      set       ! make note of count exhaustion
        case vec2; but 16       ! count   0
        case vec1               ! count = 0
      tes

 .=2267; d←rdf+b   ; r[df]  ←d; cksum   ! word
 .=2367; d←rdf+b.cl; r[df].l←d; cksum   ! even byte
 .=2377; d←rdf+b.ch; r[df].l←d; cksum   ! odd byte
   end ! checksum

        Illegal                                 ! 06 - undefined


        Illegal                                 ! 07 - undefined

  tes   ! opcode decode

vec6:   .=2007; d←rdf; prop; rdf←d
        p3; ir←ir               ! restore old ir
        d←count; but 16         ! update the count
        rdf←d  ; goto 16        ! return to the ROM
```