Computer Science Department
114 Lind Hall
Institute of Technology
University of Minnesota
Minneapolis, Minnesota 55455

# PDP-11/40E Microcode Simulator Primer

by

H. K. Berg and B. E. Blasing

Technical Report 78-10
July 1978

Cover design courtesy of Ruth and Jay Leavitt

# PDP-11/40E Microcode Simulator Primer

by

Helmut K. Berg and Bradford E. Blasing
Department of Computer Science
University of Minnesota

## Abstract

This report is an introductory guide to the use of the PDP-11/40E microcode simulator. It is intended to familiarize new users with the microcode simulator as one of the microprogram development aids in the microprogramming laboratory, and as a reference for advanced users. The operation of the microsimulator and its interface with the MICRO/40 assembler and the UNIX operating system are described. The simulator commands and their use are presented in the form of a tutorial. Sufficient detail on the PDP-11/40E hardware and the inconsistencies of its model in the simulator software is provided such as to identify the limitiations of the simulator's error detection and location capabilities. A classification of the errors which are detected by the simulator and guidelines for the interpretation of the associated error messages complement the information needed for the day-to-day use of the microcode simulator. The concept and use of the microcode simulator is demonstrated by a complete simulator terminal session.

## 1. Introduction

The microcode simulator described in this report allows interactive testing of PDP-11/40E user microprograms written in the MICRO/40 assembly language [1]. The microassembler and simulator were developed at Carnegie-Mellon University, Department of Computer Science, to run as cross assembler/simulator on a PDP-10 computer. The simulator version we refer to in this report is a PDP-11 version of the original simulator that was written at the Technical University Berlin, Institut für Softwaretechnik und Theoretische Informatik, Fachgebiet Betriebssysteme. It runs together with the MICRO/40 assembler on a PDP-11/40 under the UNIX operating system [2,3]. The simulator is written in the "C" programming language [4], except for some UNIX-assembler procedures.

The PDP-11/40E was developed at Carnegie-Mellon University [1,5]. It is a standard PDP-11/40 computer that has been extended by the following hardware features:

- 1K 80-bit words of random access (RAM) control store for storing user microprograms.
- 32 80-bit words of read-only (PROM) control store for bootstrapping microprograms.
- a 16-word stack for temporary data storage.
- a shift and mask unit and a carry control unit which extend the data manipulation capabilities of the basic PDP-11/40 processor.

The 3-Rivers Computer Corporation offers these hardware accessories as a writable control store option (WCS 11/40) for the PDP-11/40. The design of this extension allows user microprograms access to all functional hardware units and data paths in the basic PDP-11/40 processor and in the WCS 11/40. Register-transfer block diagrams of the PDP-11/40 processor, WCS 11/40, and their mutual interfaces are given in the appendix. Introductions to the microprogramming of the PDP-11/40E and the MICRO/40 assembler are given in [1, 6, 7].

This report is intended to provide an introductory guide to the use of the microsimulator as one of several microprogram debugging aids for the PDP-11/40E [8, 9]. Therefore, we first discuss (section 2) the position of microprogram simulation among other microcode validation techniques. The operation of the PDP-11/40E microcode simulator and its interface with MICRO/40 are described in section 3. To proceed, in section 4, the simulator commands are specified and their use is demonstrated by examples. Then, the simulator

error messages are discribed, and guidelines for their interpretation are given in section 5. Following comments on the implementation of the simulator in section 6, we show, in section 7, a complete example of the use of the simulator.

## 2. Microcode Simulators

The basic approaches to program validation are formal correctness proofs and testing. Proofs of formal correctness attempt to show the absence of errors, whereas testing is only capable of showing the presence of errors, but not their absence. Both methods must be supported by debugging techniques which aid the location and correction of errors. The lower complexity of operations and data structures affected by microinstructions and the small size of microprograms as compared to software programs further formal correctness proofs of microcode. However, the fact that microprograms affect only a small and well-defined set of resources and data items has also led to microcode testing and debugging techniques which are not practical for software. Microcode simulation is such a testing technique.

A microcode simulator is a software program that simulates the execution of microoperations in the functional hardware units and data paths of a processor. Such simulation programs usually allow the tester to interactively trace the execution of individual microinstructions in a microprogram. Furthermore, facilities are provided for locating and correcting errors by examining and changing register contents at specified breakpoints in the simulation run. The PDP-11/40E microcode simulator includes all these standard features.

A microsimulator represents one of a collection of utilities for microprogram construction, testing, debugging, and maintenance in a microprogramming support system. A typical microprogramming support system includes a microassembler, a microsimulator, and a microprogram loader. More sophisticated support systems may also include test set generators, external hardware accessories for microprogram instrumentation, and microprogram verification systems, etc. The loader in such a system usually transforms the output from the microassembler such that it can directly be interpreted by the microsimulator, and loads the binary version of the assembled microcode into the control store. In our microprogramming support system for the PDP-11/40E, the MICRO/40 assembler generates files to be used by the microsimulator and a separate microprogram load and test system,

SMILE [10], [8].

For program testing, in general, we can distinguish between static and dynamic variants. Static testing, via program analysis, has not been applied to microprogramming. Dynamic microprogram testing, i.e., testing via microprogram execution, can be categorized as soft (off-line) testing or hard (on-line) testing. A microcode simulator is a typical off-line test system. Generally, only static programming errors can be detected by simulator testing. The ability to discover dynamic timing errors with a simulator depends on how closely the simulator implementation reflects the microarchitectural characteristics of the machine. Debugging of dynamic errors may be supported by microinstruction formats which make microoperation timing conflicts detectable through microcode examination in the microassembler. However, the detection of dynamic errors usually requires on-line test procedures. Typical on-line microprogram testers are interactive debuggers and special hardware accessories for the instrumentation of microprogram executions. An interactive debugger essentially allows the standard simulator commands to be performed with respect to the execution of microprograms in the physical machine. A straightforward implementation of a PDP-11/40E debugger is not possible, as important resources such as the instruction register (IR), the bus address register (BA), etc. are not accessible. Therefore, the use of a logic state analyzer as an on-line microprogram debugging aid for the PDP-11/40E has been investigated [9].

## 3. Microcode Simulator Operation

In this section, we discuss the basic operation modes of the micro-simulator. For a detailed description of the microassembler operation, the reader is referred to [7]. Throughout this report, we refer to the example microprogram,[1)] called fastc.mic, shown in Fig.1. The microprogram, fastc.mic, implememts two machine language subroutines that handle the environment switch for subroutine calls in the "C" programming language of UNIX. It saves and restores registers that are used for parameter passing in subroutine calls. Further details of fastc.mic will be introduced as needed.

---

1) This microprogram was developed by K.Bullis, J.Bjoin, and T.Lunzer as a course project for (H.K.Berg) CSci 5299, Microprogramming, Winter Quarter 1978.

```
require defs.mic

begin noop
.=2001; d_210; b_d
d_rir-b !compare instruction
skipzero
d_211; b_d

set

  start          !check for other instr
    d_rir-b
    skipzero
    noop

    set

        goto 150

      start      !211 instr
        d_r5     !r1<-r5
        r1_d
        d,ba_r1-2; r1_d !pop r4
        dati; clkoff
        r4_unibus
        ba,d_r1-2; r1_d !pop r3
        dati; clkoff
        r3_unibus
        ba,d_r1-2; r1_d !pop r2
        dati; clkoff
        r2_unibus
        ba,d_r5 !sp<-r5
        r6_d; dati; clkoff
        r5_unibus        !r5<-(sp)+
        d_r6+2; r6_d
        ba_r6; dati      !rts pc
        d_r6+2; r6_d; clkoff
        r7_unibus; but 16
        goto 16
      end

    tes
  end

start          !210 instr
      d,ba_r6-2; r6_d !push r5
      d_r5; dato; clkoff
      d_r7; p3         !r5<-r7
      r5_d
      r0_d     !r0<-r5
      d_r6     !r5<-r6
      r5_d
      d,ba_r6-2; r6_d !push r4
      d_r4; dato; clkoff
      d,ba_r6-2; r6_d !push r3
      d_r3; dato; clkoff
      d,ba_r6-2; r6_d !push r2
      d_r2; dato; clkoff
      d_r6-2; r6_d     !r6<-r6-2
      d_r0     !r7<-r0
      r7_d; but 16
      goto 16
  end

tes

finis
```

Figure 1: fastc.mic

The first line of fastc.mic is a command to the MICRO/40 assembler that "requires" the microcode source file defs.mic to be bound with the actual microcode of this microprogram. The file defs.mic contains standard macro definitions which have been found to be of general use. defs.mic is listed in Fig. 2. The macros in defs.mic define mnemonic names for registers, microoperations, and microinstruction addresses which can subsequently be used in fastc.mic. The primary purpose for including the given macro difinitions is to make the source microcode of fastc.mic more comprehensible.

User microcode for the PDP-11/40E is written in the MICRO/40 assembly language. Microcode source files are generated using the UNIX text editor [2]. The name of a microprogram source file must be of the form

<name>.mic,

where <name> is any legal name, e.g., fastc. The suffic "mic" indicates that the source is written in MICRO/40.

```
! 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 := bes: .=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
! ADDITIONAL MACROS
POP:=DEST=1;MSC=4$
PUSH:=DEST=1;MSC=3$
```

Figure 2: defs.mic

To assemble a microprogram source file named, <name>.mic, the command [2)]

mic[opt]<name>.mic

---

2) Throughout this text we use the following meta symbols:

- [] - encloses optional objects in command lines (i.e., 0 or 1 repetition allowed).
- {} - denotes multiple occurrences of objects in command lines (i.e., 0 or more repetitious allowed).

is typed, where opt is one of the three legal options -a, -s, or -d. If none of these options is used in the above microassembler call (i.e., mic <name>.mic), three files, <name>.lst, <name>.bin, and <name>.tab are generated. <name>. lst is a listing of the microprogram object code in the 80-bit microinstruction format, followed by a list of mnemonic addresses and associated control store addresses. Fig. 3 shows this listing for the example microprogram fastc.mic. The file <name>.bin is the binary version of the assembled microcode which is loaded into the writable control store. <name>.tab is discussed below.

The options in the microassembler call instruct MICRO/40 to generate the following output.

-a The microprogram source is assembled and a pseudo-readable form of the object microcode is stroed in a file called <name>.ass. This file is in UNIX assembler format. From this file, the binary version of the assembled microcode, <name>.bin, can be generated using the UNIX assembler.

-d The microprogram source is assembled and a pseudo-readable form of the object microcode is stored in a file called <name>.dec. This file is in DEC assembler format, so that the binary version of the assembled microcode can be generated using the DEC PDP-11 assembler.

-s The microprogram source is assembled and the simulator is called, if no assembly errors occured.

Figure 3: fastc.lst

| inst | adr | emit | scom | ppe | cp | sc | rm | lm | dest | msc | xupf | clk | ir | wr | b | d | ba | bus | dad | sps | alu | bc | bm | dm | bam | ubf | srx | rif |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| beginnoop | 2000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2001 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| .=2001;d_210;b_d | 2001 | 210 | 0 | 0 | 0 | 0 | 10 | 10 | 0 | 1 | 2006 | 6 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 |
| d_rir-b | 2002 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2011 | 4 | 0 | 0 | 0 | 1 | 0 | 0 | 10 | 0 | 6 | 0 | 0 | 0 | 0 | 0 | 1 | 13 |
| d;ba_r6-2;r6_d!pu | 2003 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2035 | 6 | 0 | 3 | 0 | 1 | 1 | 0 | 10 | 0 | 6 | 2 | 17 | 2 | 0 | 0 | 1 | 6 |
| goto150 | 2004 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 150 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| d_r5!r1<-r5 | 2005 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2013 | 4 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 5 |
| d_rir-b!comparein | 2006 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2007 | 4 | 0 | 0 | 0 | 1 | 0 | 0 | 10 | 0 | 6 | 0 | 0 | 0 | 0 | 0 | 1 | 13 |
| skipzero | 2007 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2010 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 12 | 0 | 0 |
| d_211;b_d | 2010 | 211 | 0 | 0 | 0 | 0 | 10 | 11 | 0 | 1 | 2002 | 6 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 |
| skipzero | 2011 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2012 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 12 | 0 | 0 |
| noop | 2012 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2004 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| r1_d | 2013 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2014 | 2 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 1 | 1 |
| d;ba_r1-2;r1_d!po | 2014 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2015 | 6 | 0 | 3 | 0 | 1 | 1 | 0 | 10 | 0 | 6 | 2 | 17 | 2 | 0 | 0 | 1 | 1 |
| dati;clkoff | 2015 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2016 | 3 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| r4_unibus | 2016 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2017 | 2 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 4 |
| ba;d_r1-2;r1_d!po | 2017 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2020 | 6 | 0 | 3 | 0 | 1 | 1 | 0 | 10 | 0 | 6 | 2 | 17 | 2 | 0 | 0 | 1 | 1 |
| dati;clkoff | 2020 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2021 | 3 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| r3_unibus | 2021 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2022 | 2 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 3 |
| ba;d_r1-2;r1_d!po | 2022 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2023 | 6 | 0 | 3 | 0 | 1 | 1 | 0 | 10 | 0 | 6 | 2 | 17 | 2 | 0 | 0 | 1 | 1 |
| dati;clkoff | 2023 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2024 | 3 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| r2_unibus | 2024 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2025 | 2 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 2 |
| ba;d_r5!sp<-r5 | 2025 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2026 | 4 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 5 |
| r6_d;dati;clkoff | 2026 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2027 | 3 | 0 | 3 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 1 | 6 |
| r5_unibus!r5<-(sp | 2027 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2030 | 2 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 5 |
| d_r6+2;r6_d | 2030 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2031 | 6 | 0 | 3 | 0 | 1 | 0 | 0 | 0 | 0 | 11 | 2 | 17 | 2 | 0 | 0 | 1 | 6 |
| ba_r6;dati!rtspc | 2031 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2032 | 2 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 6 |
| d_r6+2;r6_d;clkof | 2032 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2033 | 7 | 0 | 3 | 0 | 1 | 0 | 0 | 0 | 0 | 11 | 2 | 17 | 2 | 0 | 0 | 1 | 6 |
| r7_unibus;but16 | 2033 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2034 | 2 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 16 | 1 | 7 |
| goto16 | 2034 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 16 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| d_r5;dato;clkoff | 2035 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2036 | 5 | 0 | 0 | 0 | 1 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 5 |
| d_r7;p3!r5<-r7 | 2036 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2037 | 6 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 7 |
| r5_d | 2037 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2040 | 2 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 1 | 5 |
| r0_d!r0<-r5 | 2040 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2041 | 2 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 1 | 0 |
| d_r6!r5<-r6 | 2041 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2042 | 4 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 6 |
| r5_d | 2042 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2043 | 2 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 1 | 5 |
| d;ba_r6-2;r6_d!pu | 2043 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2044 | 6 | 0 | 3 | 0 | 1 | 1 | 0 | 10 | 0 | 6 | 2 | 17 | 2 | 0 | 0 | 1 | 6 |
| d_r4;dato;clkoff | 2044 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2045 | 5 | 0 | 0 | 0 | 1 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 4 |
| d;ba_r6-2;r6_d!pu | 2045 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2046 | 6 | 0 | 3 | 0 | 1 | 1 | 0 | 10 | 0 | 6 | 2 | 17 | 2 | 0 | 0 | 1 | 6 |
| d_r3;dato;clkoff | 2046 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2047 | 5 | 0 | 0 | 0 | 1 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 3 |
| d;ba_r6-2;r6_d!pu | 2047 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2050 | 6 | 0 | 3 | 0 | 1 | 1 | 0 | 10 | 0 | 6 | 2 | 17 | 2 | 0 | 0 | 1 | 6 |
| d_r2;dato;clkoff | 2050 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2051 | 5 | 0 | 0 | 0 | 1 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 |
| d_r6-2;r6_d!r6<-r | 2051 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2052 | 6 | 0 | 3 | 0 | 1 | 0 | 0 | 10 | 0 | 6 | 2 | 17 | 2 | 0 | 0 | 1 | 6 |
| d_r0!r7<-r0 | 2052 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2053 | 4 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| r7_d;but16 | 2053 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2054 | 2 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 16 | 1 | 7 |
| goto16 | 2054 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 16 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

labels and addresses
====================

| | |
|---|---|
| bes | 2000 |

When the microassembler is called with the "-s" option, it generates the following two files to be used by the microsimulator.

<name>.bin

The binary version of the assembled microcode is generated from <name>.ass, using the UNIX assembler as a post-processor of MICRO/40.

<name>.tab

This file is generated specifically for the simulator. It contains information used by the simulator for interperting simulator commands. This information includes field names, field bounds, lower and upper limits (control store addresses) of the microcode generated, register names, macro names, labels, etc. From this file, the simulator generates a symbol table which is used to interpret simulator commands. As a result, simulator commands can contain any symbol string that is recognized by the microassembler, including names of user-defined macros.

The call of MICRO/40 with the "-s" option invokes the microsimulator after assembling the microprogram. This invocation is useful, when assembled microcode is to be tested immediately after assembling. For instance, when changes are made to a microprogram, the effect of these changes can be observed by simulating the reassembled microcode. Alternatively, the microsimulator can be called directly on a pre-assembled microprogram, using the command

sim <name>.bin <name>.tab .

This invocation allows the microprogrammer to simulate his microprogram in several terminal sessions without reassembling the source microcode.

When the simulator is invoked, it goes into command mode, which is indicated by its prompting with ">" on the DEC-writer. In this mode, any of the commands discussed in section 4 are always legal. If a command is given which conflicts with a previous command, the microsimulator obeys the more recent command. The simulator leaves the command mode, when a command is issued that effects the execution of a microprogram in the simulated PDP-11/40E hardware (G or S command). It returns to command mode whenever the execution of a microprogram is halted. A return to command mode may be caused by setting breakpoints in the microprogram under execution, encountering certain errors, or an user interrupt generated by pressing the delete key at the DEC-writer. User interrupts may be generated at any time during microprogram execution and cause the simulator to enter command mode after the execution of the current microinstruction. A simulation run is terminated, i.e., control is returned to the UNIX operating system, by the "exit" command.

The control store address space of the microsimulator is limited to the writable control (RAM) and the bootstrap control store (PROM) in the WCS 11/40. That is, any control store address in the range $[0_8:377_8]$ (address space of the PDP-11/40 emulator ROM) causes an address error in the microcode simulation run. Thus, for each control transfer from the WCS 11/40 control store to the standard PDP-11/40 ROM, the simulator automatically interrupts the simulation run and returns to command mode.

Besides commands that allow for the location of errors, the microsimulator provides facilities for microcode correction (modification). To this end, the binary representation of the microcode as used by the simulator can be changed by reassigning the content of individual 16-bit main memory words (cf. SET command). Note, however, that this facility supports only minor changes in existing microinstructions which may immediately be tested during microcode simulation. Microcode changes as carried out with the simulator do not affect the microcode to be loaded into the WCS 11/40 control store, i.e., such changes do not modify the object microcode in the file <name>.bin. Hence, to retain changes made during microcode simulation requires modification and reassembling of the source microcode in <name>.mic.

## 4. Simulator Commands

In this section, we discuss the simulator commands and give examples of their typical application. The simulator commands can be divided into two groups, namely simulation control commands and input/output control commands. The first group of commands allows the user to specify control paths and modes for the simulation of a given microprogram. The second group of commands provides means to investigate the result of simulation and to initialize the simulated hardware of the PDP-11/40E.

Simulator commands are written, one per line, at the DEC-writer. The syntax of the command lines is

<command name>[<identifier>]{<delimiter><identifier>}.

Delimiters are either the symbol "=" or the symbol ",". Identifiers may represent <number>, <register>, <label>, or <name>. <number> can be any integer in the range [-32768:65535]. Numbers greater than 32767 are assumed to be represented in a 16-bit word without sign bit. All numbers are assumed to be octal, unless they contain the digits 8 or 9, or a trailing decimal point in which case they are assumed to be decimal. <register>, <label>, and <name> stand for any string that can be identified via the simulator symbol table. <register> must represent a register name, including user-defined

macros. <label> may represent a mnemonic label or a number identifying a microinstruction address in the control store. <name> stands for any string identifying a register, a label (i.e., a microinstruction), a macro, or a UNIX file. The legitimate identifiers for registers, mnemonic labels, and macro are defined by the source microcode in <name>.mic and the associated macro definitions (cf. Fig.1 and Fig.2). Numbers identifying microinstruction correspond with the control store addresses as assigned in the file <name>.lst (cf. Fig.3).

## 4.1 Simulation Control Commands

GO

g [<label>]

This command starts or continues execution (simulation) of the microprogram. The specification of a label is optional. If a G command is given without a label, before any microinstruction has been executed, execution begins at location 2000 (first word in the writable control store). If a G command without a label is given, after some microinstructions have been executed, simulation continues where the simulation was last interrupted. If a label is specified in a G command, execution starts with the microinstruction stored at the associated control store location. For example, the command

g begin

starts simulation of the microprogram fastc.mic at location 2000 (cf. Fig.1 and Fig.2). The command

g 2035

begins execution with the microinstruction

d_r5;dato;clkoff

(cf. Fig.3).

When the simulation is invoked by a G command, the simulator leaves command mode and executes the microprogram starting from the defined location. It returns to command mode when an error occurs, a previously set breakpoint is reached, or an interrupt is generated by the user. For example, when the command, g begin, is given and a machine instruction other than $210_8$ or $211_8$ executed, the microinstruction, goto 150, is encountered in fastc.mic and hence, an address error occurs. At this point the simulator prints the error message

150: address out of bounds ,

before returning to command mode (i.e., prompting with ">"). For the command, g 2035, the exit from the WCS 11/40 control store address space is taken at control store location 2054 and the error message,

16: address out of bounds ,

is printed.

STEP

s [<label>][,<number>]

This command starts or continues execution (simulation) of the microprogram in a stepping mode. Analogous to the G command, execution starts at location 2000, if the S command is given without label and no microinstructions have been executed. If a label is given, the stepping begins with the microinstruction stored at the associated control store location; otherwise, the stepping continues from the point where the simulation was last halted.

The number in the S command specifies the number of microinstructions to be stepped through. If no number is given, the default '1' is assumed, and the simulator returns to command mode after the execution of a single microinstruction. While stepping through the microporgram, the control store addresses (not the mnemonic labels) of the simulated microinstructions, are printed out, before the appropriate microinstruction is executed. Therefore, this command is especially useful for examining branches in a microprogram, as a trace of all executed control store locations is generated.

For our example microprogram, fastc.mic, the initial invokation of the simulation with the S command,

s,5 ,

steps through five microinstructions generating the output,

2000
2001
2006
2007
2010
> ,

if a machine instruction other than $210_8$ is executed. The S command,

s begin,5 ,

obviously does the same, whenever it is given during the simulation process. As another example, the S command,

s 2035 ,

executes the microinstruction,

```
d_r5; dato; clkoff,
```

with the output,

```
2035
>        .
```

The S command causes the simulator to leave command mode and to step through the specified number of microinstructions, before returning the command mode. If an error occurs, a previously set breakpoint is reached, or an interrupt is generated by the user, it returns to command mode, before the specified number of microinstructions has been simulated. For example, the S command,

```
s 2000, 10
```

generates an address error at location 2004 of fastc.lst and generates the output,

```
2000
2001
2006
2007
2010
2002
2011
2012
2004
address out of bounds
>            .
```

TRACE

```
t  <name>{,<name>}
```

This command sets a trace flag on either a register, a mnemonic label, or a control store address in the microprogram. If a register is being traced, its new content, and the control store address of the microinstruction which modified it are printed after each modfication. A microinstruction is traced by printing the label or control store address as specified in the T command, each time that microinstruction is executed. Any combination and number of registers and microinstructions may be traced simultaneously by specifying a list of register names and labels in the T command.

Registers and microinstruction which are known under more than one name can be traced under only one name at a time. If an attempt is made

to trace a register or microinstruction under more than one name, only the name specified last will be used (and printed).

In the case of our example microprogram, fastc.mic, we might be interested in tracing the registers R[2], R[3], and R[4]. These registers are saved in the second segment of fastc.mic ($210_8$ instruction) by pushing them onto the stack, and restored in the first segment of fastc.mic ($211_8$ instruction) by popping them off the stack. This trace can be accomplished by the commands

```
t r2, r3, r4
> g 2013
```

If the stack contains the values 10, 11, and 12 for the registers R[2], R[3], and R[4], respectively, this commands generate the output,

```
2016: r[4] = 12
2021: r[3] = 11
2024: r[2] = 10
16:  address out of bounds
>
```

In this case, simulation was started at control store location 2013 and hence, exits at location 2034 with an address error.

To examine whether fastc.mic was invoked by a $210_8$ instruction (jump to location 16 at location 2054), a $211_8$ instruction (jump to location 16 at location 2034), or an illegal instruction (jump to location 150 at location 2004), the exits from the writable control store may be traced as follows,

```
t 2004, 2034, 2054
```

For different invokations, the simulator generates the following outputs,

```
> g 2013
2034
16: address out of bounds
> g 2035
2054
16: address out of bounds
> g 2000
2004
150:  address out of bounds
>
```

A T command may be used in stepping mode. In this case, addresses of traced microinstructions are printed twice. For example, the following commands, in which traces for microinstructions and registers are combined,

```
t r2, r3, r4, 2015, 2017
> s 2013,  8   ,
```

generate the following output

```
2013
2014
2015
2015
2016
2016:  r[4]=16
2017
2017
2020
2021
2021: r[3]=10741
2022
>              .
```

When tracing, the simulator stops every 16 lines of output and prints a "*". This measure is included since the simulator was originally designed to be used with a CRT on which only 16 lines of output can be displayed at once. The traced execution of a micro-program is resumed, when a "return" is typed.

BREAK

```
b <name>{,<name>}
```

The B command is a variant of the T command, with the difference being that it causes a break whenever a traced register is modified or a traced micoinstruction is executed. When a breakpoint is reached, the simulator prints the trace information and returns to command mode. Before the break occurs, the execution of the microinstruction broken on or in which a register being broken on is modified will be completed.

In the B command, the same restrictions for variable names and microinstruction labels apply as in the T command. Additionally, it is not possible to break and trace on the same register or microinstruction at the same time. If an attempt is made to break and trace on the same name, or more than one name was given for the same register

or microinstruction, the name specified last will be used (and printed) in the last command issued.

The following examples for the use of the B command correspond to those discussed for the T command. Note the difference in the output for the B commands and T commands.

```
b r2, r3, r4
> g 2013
break 2016; r[4] = 12
> g
break 2021: r[3] = 11
> g
break 2024: r[2] = 10
> g
16:  address out of bounds
>

b 2004, 2034, 2054
> g 2013
break: 2034
> g
16: address out of bounds
> g 2035
break: 2054
> g
16: address out of bounds
> g 2000
break: 2004
> g
150: address out of bounds
>
```

The trace contained in a B command is continued when a break occurs by giving a G command or a S command. However, in contrast to the T command, the fact that the simulator returns to command mode on a break allows the examination of registers which are not specified in the B command (cf. input/output control commands). For the commands,

```
b r2, r3, r4, 2015, 2017
> s 2013, 8   ,
```

we obtaine the output,

```
2013
2014
break: 2015
2015
> g
break 2016: r[4] = 16
2016
> g
break: 2017
2017
> g
2020
break 2021: r[3] = 10741
2021
> g
2022
>
```

REMOVE

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

This command removes the specified registers or microinstructions from the break or trace list of the currently active B commands and T commands. That is, in subsequent invokations of the simulation, the registers or microinstructions specified in the R command are no longer traced or broken on. If no names are given in the R command, all tracing and breaking is stopped.

It is not necessary by specify the same register names and microinstruction labels or addresses in the R command that were given in the B commands or T commands to be removed. Synonyms are allowed, as the simulator responds to the registers and microinstruction addresses as defined in the symbol table, not to be the names which provide entries into the symbol table.

Given the T command,

t 2004, 2034, 2054

and the B command,

b r2, r3, r4, r6.

The command

r 2034, 2054, rsp

reduces the T command to a trace of the microinstruction stored at location 2004 (i.e., goto 150, cf. Fig.2) and restricts the B command to a break on modifications of registers R[2], R[3], and R[4]. The name "rsp" is a synonym for r6, and is defined as a macro in the file defs.mic (cf. Fig.2). Note that although the stack pointer, R[6], was specified by the name r6 in the B command, it can be removed from the break list using the macro, rsp, in the R command.

EXECUTE

X <filename>

This command allows the user to specify a UNIX file from which simulator commands are read and executed. Any legal UNIX file name can be used to name a simulator command file. The commands specified in a simulator command file must not invoke the simulator to leave command mode. Furthermore, recursive calls of the X command are not allowed. Therefore, the simulator commands G, S, E, P (cf. subsection 4.2), and X are illegal commands when read from a simulator command file. Any other legal command can be executed.

The X command is useful to define a standard set of frequently executed commands. For example, a standard set of T commands and B commands may be stored in a UNIX file, debug, and may be executed, whenever the simulator output deviates from the expected result. In the case of our example microprogram, fastc.mic, the file, debug, may contain the simulator commands

t 2004, 2034, 2054 and b r2, r3, r4, r6.

Then the command

X debug

corresponds to issuing the above T and B commands separately, i.e., for the execution of a $210_8$ instruction, we obtain,

```
X debug
> g
break 2003: r[6] = 177776
> g
break 2043: r[6] = 177774
> g
break 2045: r[6] = 177772
> g
break 2047: r[6] = 177770
> g
break 2051: r[6] = 177766
> g
2054
16: address out of bounds
>
```

EXIT

e

The E command returns control to the UNIX operating system. It is the only simulator command that allows the termination of a simulation run. As the E command has no argument it is not possible to specify a termination condition that is tested by the simulator. Hence, a simulation run can only be terminated, when the simulator returns to command mode. Note that the occurrence of an error, breakpoint, or user-generated interrupt are the only means that cause the simulator to return to command mode.

## 4.2 Input/Output Control Commands

GET

=<name>

This command allows the current content of registers, locations of the simulated core (cf. LOAD command), or control store locations to be printed. Registers may be specified by any name that is defined in the simulator symbol table. Register values are printed as octal numbers. Locations of the simulated core are specified by octal numbers in the range $0000_8$ to $1776_8$ (1K of memory words). The 16-bit content of core locations is printed in octal. Control store locations can only be specified by numbers representing control store addresses. The contents

of control store locations, i.e., 80-bit microinstructions, are printed in the form of five separate 16-bit fields. That is, five separate GET commands must be given to output the octal representation of a complete 80-bit microinstruction. Therefore, the addresses that identify a 16-bit field in the writable control store are specified as 16-bit addresses with the address format (cf.[6]) discussed below.

Fig.4 illustrates the PDP-11/40E control store address space. In the 16-bit control store address representation, A, the 10-bit

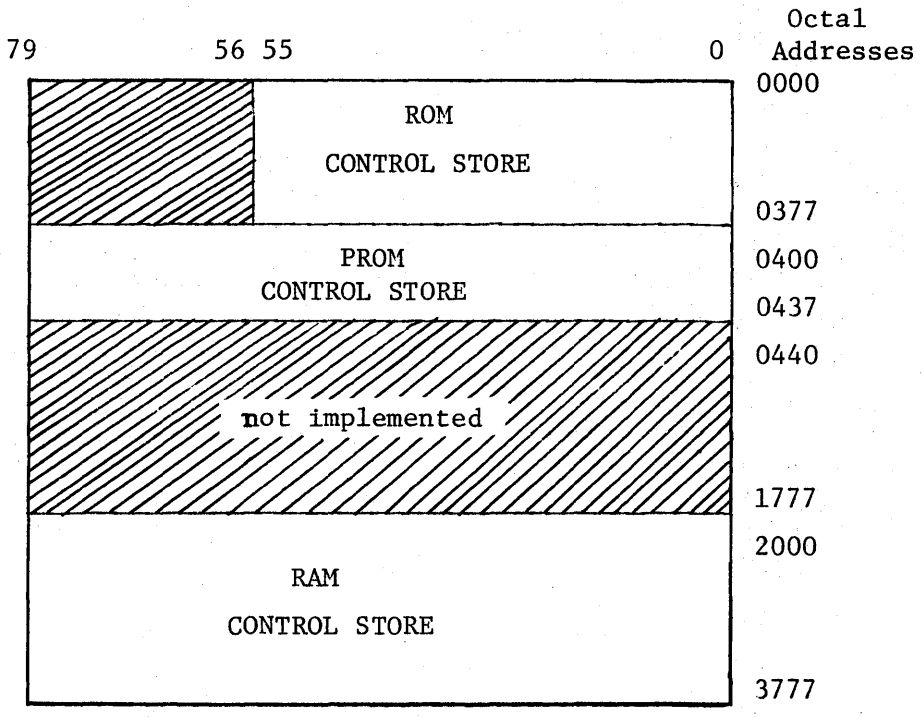



Figure 4: Extended Control Store Address Space

RAM addresses are contained in bits A<12:3>. Bits A<14,2:1> identify one of the five 16-bit fields in the 80-bit microinstruction. The remaining bits, A<15,13,0>, are ignored. Hence, the RAM address map as shown in Fig.5 is obtained.

| A<14,2:1> | | | | | |
|---|---|---|---|---|---|
| 100 | 011 | 010 | 001 | 000 | A<12:3> |
| 60000 | 20006 | 20004 | 20002 | 20000 | 0000 |
| 60010 | 20016 | 20014 | 20012 | 20010 | 0001 |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| 77760 | 37766 | 37764 | 37762 | 37760 | 1776 |
| 77770 | 37776 | 37774 | 37772 | 37770 | 1777 |

Figure 5: RAM Address Map.

The field addresses, FA, are constructed as follows:
FA= A<14>, 1, A<12>, A<11:9>, A<8:6>, A<5:3>, A<2:1>, 0.

The five 16-bit fields that compose an 80-bit microinstruction are depicted in Fig.6 (cf. Fig.3).

Field 0: <A 14,2:1> = 000

| 15–14 | 13–12 | 11–8 | 7–6 | 5–4 | 3–0 |
|---|---|---|---|---|---|
| SDM | SBAM | UBF | SRS | SRD | SRBA | SRI | RIF |

Field 1: A<14,2:1> = 001

| 15–13 | 12–11 | 10–8 | 7–4 | 3–2 | 1–0 |
|---|---|---|---|---|---|
| SPS | SALUM | SALU | SBC | SBMH | SBML |

Field 2: A<14,2:1> = 010

| 15–14 | 13–12 | 11 | 10 | 9 | 8–7 | 6 | 5 | 4–3 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| CLKL | CLKOFF | CLKIR | WRH | WRL | CLKB | CLKD | CLKBA | CBUS | BGBUS | DAD |

Figure 6: continued.

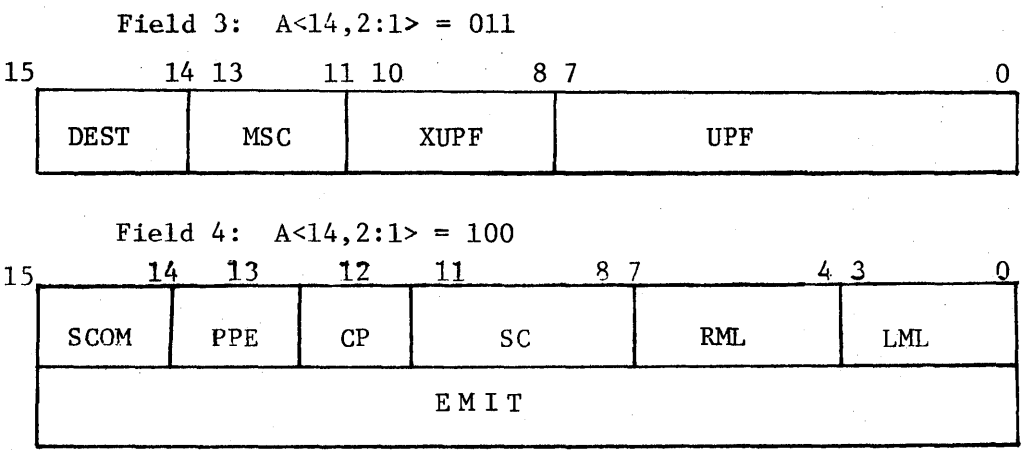



Figure 6: 16-bit Field Assignments in 80-bit Microinstructions.

The GET command is useful to investigate the processor state, when an error or a user-generated interrupt caused the simulator to return to command mode. The following examples refer to our example microprogram, fastc.mic.

To demonstrate the use of the GET command to output RAM contents, we give give the command sequence for printing the content of RAM location 2000 (cf. Fig.3):

```
= 20000
20000: 0
>= 20002
20002: 0
>= 20004
20004: 40000
>= 20006
20006: 2376
>= 60000
60000: 0
>
```

Note that field 3 in the microinstruction at location 2000, <DEST,MSC, XUPF,UPF> = 002001, is represented by the value 002376. This is due to the fact that the content of UPF (cf. Fig.6) is complemented in its representation. That is, the correspondence between the representation of field 3 in <name>.lst, aaaxxx, and the representation of field 3 in the output of the GET command, aaayyy, is established by the relationship,

aaayyy = aaa377 - 000xxx.

An example for the output of the contents of simulated core locations with the GET command is given below. For this example, core locations 0, 2, and 4 are set to the values $1000_8$, $1001_8$, and $1002_8$, respectively:

```
= 0
0: 1000
>= 2
2: 1001
>= 4
4: 1002
>      .
```

The output of register contents is handled as follows. Registers R[2], R[3], and R[4] are set to the values 10, 11, and 12, respectively.

```
= r2
r2: 10
>= r3
r3: 11
>= r4
r4: 12
>      .
```

SET

_<name> = <number>

This command allows the user to set the content of registers, locations of the simulated core (cf. LOAD command), or control store locations to the given number. The identification of carries and the specification of carrier contents is analogous to the GET command.

Changes of register, core location, or control store location contents as specified in the SET command are made immediately, without the simulator leaving command mode. The old carrier contents are lost. Note that the SET command allows the user to modify (correct) microcode during the simulation run (cf. section 3). The following examples for the use of the SET command correspond to the examples given for the GET command.

Set control store location.

```
_ 20000 = 0
>_ 20002 = 0
>_ 20004 = 40000
>_ 20006 = 2376
>_ 60000 = 0
>
```

Set core locations.

```
_ 0 = 1000
>_ 2 = 1001
>_ 4 = 1002
>
```

Set register contents.

```
_ r2 = 10
>_ r3 = 11
>_ r4 = 12
>
```

OPCODE

o<number>

This command allows the user to specify an op-code to be interpreted by the microprogram under investigation. In the real machine, control is transferred to the WCS 11/40, when an unused PDP-11/40 op-code is encountered (cf. [6]). At this point, the machine instruction with the unused PDP-11/40 op-code is placed into register R[13], the instruction register IR, and the B register. The bootstrap microcode in the WCS 11/40 PROM transfers the machine instruction also to the top of the WCS 11/40 stack (TOS). Therefore, the 0 command sets the registers R[13], IR, B, and TOS to the specified octal number:

```
o 211
>= 6
b: 211
>= ir
ir: 211
>= tos
```

```
tos: 211
>= r[13]
r[13]: 211
>
```

The O command is desirable for the test of user microprograms which implement several machine instructions. In our example microprogram, fastc.mic, the $210_8$ and $211_8$ instructions may be tested by giving the commands

o 210

and o 211

prior to two separate simulation runs

INITIALIZE

i

This command reinitializes the simulator. Its execution is equivalent to exiting and re-invoking the simulator. That is, all PDP-11/40E registers are set to 0, the simulated core is set to 0, and the standard bootstrap PROM and the user microcode are reread. The simulator clock (cf. C command) which records the simulated execution time is reset. All simulation control tables are reinitialized such that all trace and break information is lost. Finally, the state of the simulator is reset such that G or S commands result in an original entry (location 2000) into the microprogram.

The I command is primarily used, when several simulation runs for independent microprograms are to be carried out in a single terminal session. Note, however, that the I command is not applicable to the initialization of a simulation run for the example microprogram, fastc.mic, with a $211_8$ instruction that follows a simulation run of the same microprogram with a $210_8$ instruction. This is due to the fact that the microcode segments for the $210_8$ and $211_8$ instructions are related via processor registers and the locations in the simulated core that constitute the stack. However, if a separate simulation run is to be carried out to test the instruction decoding in fastc.mic for an unrelated, illegal op-code other than $210_8$ or $211_8$, say $212_8$, the I command may be used to reinitialize the simulator.

LOAD

l <filename>

The L command loads the simulated PDP-11/40 core (not the control store) with values from the specified file. The file is a UNIX text file and can have any legal UNIX file name.

The simulated PDP-11/40 core has a size of 1K of 16-bit words. All core addresses are absolute and in the range $0000_8$ to $1776_8$. Each line of the UNIX text file to be loaded into the simulated core must have the following format

[<number>:]<number> .

The first number, if present, is the word address in core to be loaded. (This number must be even as it is a word address.) The second 6-digit octal number is the 16-bit word to be loaded.

If the first line of the file has no address, loading starts at location $0000_8$. If no address is provided for other lines in the file, the core location loaded is the location loaded last plus 2. Loading ends at the end of file mark in the UNIX text file.

The L command allows the user to initialize the environment for microprograms which reference data in main memory or execute a sequence machine language instructions from main memory. For the example micro-program, fastc.mic, we may generate the following file, called testcore,

```
70: 10
11
12
15
```

which contains the values 10, 11, 12, and 15 for the core locations 70, 72, 74, and 76, respectively. With the subsequently given initialization:

```
_rsp = 66
>_ r5 = 76
>1 testcore
>           ,
```

we obtain the following results in a simulation run:

```
= 70
70: 10
>= 72
72: 11
>_ r2=0
```

```
>_r3 = 0
>_r4 = 0
>o 211
> g begin
16: address out of bounds
>= r2
r2: 10
>= r3
r3: 11
>= r4
r4: 12
>= r5
r5: 15
>= rsp
rsp: 102
>= 70
70: 10
>= 72
72: 11
```

PROM LOAD

P <filename>

The P command provides a means for changing the 32 80-bit words of WCS 11/40 bootstrap PROM control store. Normally, the internal simulator PROM is loaded with the standard WCS 11/40 bootstrap PROM [1,6], when the simulator is initiated. The P commands reloads the simulator PROM (not the physical WCS 11/40 PROM) from the specified file. The file must be the binary version of a microprogram as generated by the MICRO/40 assembler. That is, its name is of the form <name>.bin. The dimension of PROMs to be loaded with the P command is limited to 32 80-bit micro-instructions.

The P command allows the user to test alternative microprogram bootstrap sequences. For the test of a microprogram bootstrap, boottest.bin, the simulator PROM would be loaded by the command

```
p boottest.bin
>
```

CLOCK

c

This command allows the user to inquire statistics about the execution time of the simulated microprogram. It prints two decimal values representing nanoseconds. The first of these values is the total simulated time since the simulator was invoked by the MICRO/40 assembler or since the last I command. The second value is the simulated time since the last C command.

The simulated time T for each microinstruction [1] is calculated by

$$T = T_e + T_m + T_r \quad ,$$

where $T_e$= execution time, $T_m$= memory access time, and $T_r$= regeneration time. The execution time $T_e$ for synchronous operations corresponds to the processor clock length specified in the microinstruction.

$$T_e = \begin{cases} 140 \text{ ns for P1} \\ 200 \text{ ns for P2} \\ 300 \text{ ns for P3} \end{cases}$$

The memory access time $T_m$ and the regeneration time $T_r$ may only have non-zero values, if the microinstruction performs a CLKOFF following a main memory access.

$$T_m = \begin{cases} 500-(t_{CLKOFF}-t_{MA}) \text{ for } (t_{CLKOFF}-t_{MA}) < 500 \\ 0 \end{cases}$$

$t_{CLKOFF}$ is the time when the CLKOFF is performed in the microinstruction.

$t_{MA}$ is the time when the memory access began.

$$T_r = \begin{cases} 900-(t_{CLKOFF}-t_{TMA}) \text{ for } (t_{CLKOFF}-t_{TMA}) < 900 \\ 0 \qquad \text{otherwise} \end{cases}$$

$t_{CLKOFF}$ is the time when the CLKOFF is performed in the microinstruction.

$t_{TMA}$ is the time when the first of the two most recent memory accesses began.

The C command is useful to take approximate performance measures of alternative microprograms. Additionally, the second value in the output of this command allows performance measures for microprogram segments to be taken in a single simulation run. For our example microprogram, fastc.mic, the following simulated times were obtained.

```
_r2 = 10
>_ r3 = 11
>_ r4 = 12
>_ r5 = 16
>_ rsp = 100
> o 210
> c
0. 0.      (simulated time before execution)
> g begin
16:  address out of bounds
> c
6780. 6780.  (simulated time after execution of 210)
> = rsp
rsp: 66
> o 211
> g begin
16:  address out of bounds
> c
14120.  7340. (simulated time for execution of 210
>              and 211 followed by simulated time
               for execution of 211)
```

## 5. Simulator Error Messages

Errors detected by the simulator may be grouped into execution errors and internal errors. Additionally, the simulator issues warnings. For these groups of simulator error messages, the following error types are distinguished.

Type 1: Occurs during execution (as initiated by a G or S command) and is fatal to execution, i.e., the simulator returns to command mode.

Type 2: Occurs in direct response to a command issued and inhibits the command execution.

Type 3: May occur at any point in a simulation run and is fatal to simulation, i.e., control is returned to the UNIX operating system.

Type 4: Internal error that "should not occur ", i.e., it indicates an internal failure.

Type 5: Internal error that may have been caused by the user.

Type 6: Warning.

## 5.1. Execution Errors

"illegal symbol where number expected" (Type 2)

An illegal number is encountered in a simulator command. The syntax for numbers is:

[-]<digit>{<digit>}[.] .

The number is octal, unless it includes the digits '8' or '9', or if it is postfixed with a decimal point '.'.

"recursive macro call for <name>" (Type 3)

Macros as defined in the source microcode can be interpreted by the simulator. Recursive macro expansion, however, is illegal.

"register specification without ["
"illegal general register specification for <name>" } (Type 2)
"no ] after register specification"

General register definitions must have the following syntax:

r[<spec>]

where <spec> is either

- a number is the range [0:15] or
- an assembler-defined symbol.

All other names for general registers must be defined by macros.

"don't recognize symbol" (Type 2)

The simulator command interpreter encounters an illegal or undefined symbol.

"core address out of bounds" (Type 1,2)

The core address specified in a GET or SET command is out of the range [$0000_8$: $1776_8$] of the simulated PDP-11/40 core (Type 2). This error message will also be printed when access to a non-existing core location is attempted during microcode execution (Type 1).

"address out of bounds" (Type 1,2)

The control store address specified in a G or S command is out of the range [$400_8$- $3777_8$] of the WCS 11/40 control store (Type 2). This error message will also be printed when a jump/branch to a microinstruction out of the range [$400_8$: max] is attempted (Type 1). max is the RAM control store location that contains the last micro-instruction of the user microprogram.

"symbol wrong type" (Type 2)

A symbol specified in a simulator command does not comply with the defined syntax. For example, the G and S command require a mnemonic label or an octal control store address (and not a register name).

"illegal command line" (type 2)

The simulator command interpreter cannot recognize a command, or a SET command was given without a '='.

"too many registers to trace" (Type 2)

Too many registers are specified for traces or breaks. The number of simultaneous register traces or breaks is 16.

"too many labels to trace" (Type 2)

Too many registers are specified for traces or breaks. The number of simultaneous label traces or breaks is 16.

"filename required" (Type 2)

The simulator cannot open the file associated with a P, L, or X command, because the file name is misspelled or omitted.

"too long line on this file" (Type 2)

The file specified in a P, L, or X command contains a line with more than 150 character.

"< > command ignored in command file" (Type 6)

The simulator commands G, S, E, P, and X are illegal in an X simulator command file and are ignored.

"odd address" (Type 1,6)

Odd addresses are illegal in an L command file (Type 6). When the bus address register (BA) is set to an odd value (during micro-program execution) and the controlling microinstruction does not specify a byte operation (DAD=1), the execution is aborted (Type 1).

"no branch to prom after ramread or ramwrite" (Type 1)

The timing for RAM READ/WRITE operations is such that the micro-instruction whose execution follows a RAM operation must be located in the bootstrap PROM (cf. [1,6]). The standard bootstrap PROM enforces this timing condition. However, when the P command is used, this error may occur.

"illegal ram or table address" (Type 1)

The control store address used in the execution of a RAM READ/WRITE operation is out of the range [$2000_8$: $3777_8$]. For addressing 16-bit fields of 80-bit RAM words, the (table) addresses must be in the range defined by the RAM address map given in Fig.5.

"leaving extension with unibus busy" (Type 1)

It is illegal to jump from the WCS 11/40 control store to a location in the PDP-11 standard emulator ROM while a UNIBUS operation is being performed. This restriction results from the particular UNIBUS and PDP-11/40 - WCS 11/40 interface timing conditions [1,6].

"extension switched off already" (Type 1)

The last microinstruction executed from the WCS 11/40 control store, which exists to the PDP-11/40 standard emulator ROM, uses the extension hardware. The extension hardware cannot be used at this point, as the extension is turned off, whenever the address in the microinstruction pointer field (XUPF) of the microinstruction register (U WORD) is less than $400_8$.

"msc - dest illegal combination" (Type 1)

The following combinations of the DEST/MSC fields are illegal (cf. [1], [6]):

| DEST | MSC |
|---|---|
| 10 | 000 - 011 |
| 10 | 010 - 111 |
| 11 | 011 - 100 . |

"illegal srx field" (Type 1)

Only one of the 1-bit fields, SRI, SRBA, SRD, and SRS, of the 4-bit SRX field can be set at the time (cf. [1],[6]).

"b constant doesn't exist" (Type 1)

The B constants 4, 5, and 6 are not defined and must not be specified in the SBC field (cf. [1], [6]).

"changing d register while dato active" (Type 1)

During a DATO operation, the D Register must be kept constant, as its content is gated onto the UNIBUS data lines.

"changing ba register while bus active" (Type 1)

The BA Register must be kept constant during UNIBUS operations, as

its content is gated onto the UNIBUS address lines.

"bus command or clockoff ignored" (Type 1)

UNIBUS operations, i.e., DATI and DATO, must always be followed by a 'clkoff' and a subsequent microinstruction with a P1 or P3 processor clock. Any other sequence of bus-related microoperations is illegal.

"processor stop: clockoff without preceding bus operation" (Type 1)

The execution of a 'clkoff' must be preceeded by a UNIBUS operation. Otherwise, it halts the processor, as the UNIBUS interface logic will not restart the processor clock.

"clockoff too late" (Type 1)

The execution of a 'clkoff' must follow the initiation of a UNIBUS operation within less than 500 ns.

"nonexistant ubf" (Type 1)

The microbranch conditions 13, 14, 23, and 32 are not defined and must not be specified in the UBF field (cf. [1], [6]).

"time too short for push or pop" (Type 1)

Pushing or popping of the WCS 11/40 stack requires a P2 or P3 processor clock cycle.

## 5.2 Internal Errors

"cannot open file: <name>" (Type 5)

The UNIX file specified in a P, L, or X command cannot be opened, because it either does not exist or the user has no permission to read or write it.

"read error on file: <name>"
"seek error on file: <name>"
"cannot close file: <name>"
} (Type 4)

An error occurred when a UNIX file was read.

"push: stack overflow" (Type 5)

The internal stack overflowed during simulator command parsing.

"pop: stack underflow" (Type 4)

The internal stack underflowed during simulator command parsing.

"microprogram file <name>.bin not compatible with tabfile <name>.tab" (Type 4,5)
The simulator is called with two incompatible files (Type 5). MICRO/40 transferred two incompatible files to the simulator (Type 4).

"simulator usage: <name>.bin <name>.tab" (Type 5)
The proper number of arguments (2) is not specified in the simulator call.

## 6. The Simulated PDP-11/40E

In this section we briefly discuss the PDP-11/40E and its extended 80-bit microinstruction format as modelled by the simulator software. The simulator implementation does not attempt to model the entire PDP-11/40E hardware and all possible assignments to microoperation fields in the WCS 11/40 microinstruction. Some external hardware units and the processor functions which are only useful in emulating the standard PDP-11/40 machine instruction set are omitted.

### 6.1. PDP-11/40E Hardware

The simulator models all registers and functional units in the PDP-11/40E processor. The simulation of the WCS 11/40 hardware deserves particular attention. As the WCS 11/40 registers and control stores (RAM and PROM) are not included in the PDP-11/40 UNIBUS addressing scheme, they are not accessible from the processor console or through PDP-11/40 machine language instructions, but only through WCS 11/40 microcode. Hence, using the SMILE system for microprogram loading and on-line testing [8], it is not directly possible to monitor the effect of microinstruction executions on WCS 11/40 registers. That is, the SMILE system and the microsimulator do not only complement each other with respect to the error types that may be detected, but also with respect to their capabilities to display effects of microinstruction executions, and therefore, are both necessary facilities in our microprogramming support system.

Both, the writable control store (RAM), which may also be used as data scratch pad, and the bootstrap control store (PROM) in the WCS 11/40 are modelled by the simulator. However, the PDP-11/40 standard emulator ROM is omitted from the simulator software. This organization of the simulator results from the following considerations. First, the ROM cannot be altered by the user and is considered to work correctly. Second, with the omission of the ROM from the simulator software, a jump/branch to the ROM causes an address error. The latter consideration is important, as the "exit" command does not allow the specification of a simulation termination condition, and can only be

issued, when the simulator is in command mode. Hence, a control transfer from the user microcode into the ROM automatically establishes a simulator state in which a simulation run may be terminated.

The simulator implementation does not model the PDP-11/40 UNIBUS address space, except for 1k of 16-bit main memory locations with addresses in the range $[0_8: 1776_8]$. This simulator organization is generally sufficient for two reasons. First, microprograms usually do not extensively refer to main memory. Second, references to peripherals are standardized in the UNIBUS addressing scheme and hence, can be modelled in the simulated main memory, the memory management option is not affected by the WCS 11/40 (although it may affect the WCS 11/40 [6]), and the processor console is represented by the simulator commands discussed in section 4. Note, however, that the simulator procedures 'get_adr' and 'usebus' and their calling procedures could be modified to include the UNIBUS address space, if desired.

## 6.2. WCS 11/40 Microinstruction

The following assignments to microoperations fields in the WCS 11/40 microinstruction (cf. Fig.6) are not implemented in the simulator software.

BUS

The bus field specifies and initiates UNIBUS data transfers. The assignments

BUS = 2 (await BUS BUSY)

BUS = 6 (restart on peripheral release)

are implemented as NOOPs. These definitions comply with the fact that peripherals are not modelled by the simulator.

DAD

The DAD field allows the microprogram to alter the operation of the data paths. The assignments

DAD=10 (inhibit DATO (word operation) and CLKOFF for the PDP-11/40 machine instructions BIT, CMP, or TST)

DAD=11 (inhibit DATOB (byte operation) and CLKOFF for the PDP-11/40 machine instructions BIT, MP, or TST)

are implemented as NOOPs. These definitions are due to the fact that the UNIBUS data transfers are not simulated.

SBC

The SBC field allows the selection of a B constant to be gated to the ALU B input. The generation of some of these constants is conditioned by internal processor states which are particular to the standard PDP-11/40 machine instruction set. In the simulator, all SBC assignments which test conditions always return 0 for the condition. The SBC assignments and the associated constants as used in the simulator are given below.

| SBC | B CONSTANT | |
|---|---|---|
| 1 | 1 | |
| 2 | 2 | |
| 3 | 1 | |
| 4-6 | not used | (cause Type 1 error) |
| 10 | 177570 | |
| 11 | 173374 | |
| 12 | 17 | |
| 13 | 77 | |
| 15 | 250 | |
| 17 | 4 | |
| all others | 0 | |

UBF

The UBF field specifies the branch micro test (BUT) to be performed to generate the address of the successor microinstruction by ORing the determined basic microbranch code (BUBC) into the six low-order bits of the microprogram pointer (UPP). The simulator implementation causes all UBF assignments, except UBF=12 and UBF=17, always to return 0, as the associated BUTs are particular to the standard PDP-11/40 machine instruction set. The simulator implementation of the UBF field is given below.

| UBF | BUT | BUBC |
|---|---|---|
| 12 | D=0 | 000 001 |
| 17 | IR03 | 000 001 |
| all others | - | 000 000 |

UBF=12 causes a 1 to be gated into the lowest-order bit of UPP, if the content of the D register is 0. UBF=17 causes a 1 to be gated into the lowest-order bit of UPP, if bit 3 of the instruction register is set to 1 (this bit distinguishes between direct and indirect addressing).

## 7. Terminal Session

In this section, the microcode simulator operation is demonstrated by a commented protocol of a terminal session. To this end, some minor errors have artificially be introduced into the example microprogram, fastc.mic (cf. Fig. 1 and Fig. 3). System commands and responses start at the left margin of the page. Comments are indented. Responses from the UNIX operating system end with the prompt '%'.

```
%  ed
```

The UNIX text editor [2] is used to generate a simulator command file.

```
:a
t r1
t r2
t r3
t r4
t r5
t r6
t r7
.
:
: w trace.regs
```

The generated simulator command file is written into the UNIX text file, trace.regs.

```
35
```

The number of characters in trace.regs is listed.

```
:q
```

Editing is terminated. Control is returned to the operating system.

```
%  ed
```

The UNIX text editor is reinvoked to generate another simulator command file.

```
:a
= r1
= r2
= r3
= r4
= r5
= r6
= r7
.
```

```
:w print.regs
28
:q
```

The generated simulator command file contains 28 characters and is written into the UNIX file, print.regs.

```
% ed
```

A third simulator command file is generated.

```
:a
_r1=10
_r2=12
_r3=14
_r4=16
_r5=0
_r6=1000
.
: w set.regs
43
:q
```

The generated file, set.regs, initializes the general purpose registers R[1] to R[6].

```
%  mic -s fastc.mic
```

A version of the microprogram, fastc.mic, with artificially introduced errors is assembled, and the simulator (-s) is called.

```
fastc.mic
91 lines read.
```

The MICRO/40 assembler acknowledges the acceptance of the microprogram, fastc.mic, which contains 91 lines of code. Note that no error messages are issued.

```
>
```

The simulator enters command mode after being invoked by MICRO/40.

```
>  x set.regs
>  x trace.regs
>  o 210
```

The simulator is set up for the invokation of the microprogram with a $210_8$ instruction.

```
>g
```

The simulation is started.

```
2003: r[6] = 776
2037: r[5] = 0
2042: r[5] = 776
2043: r[6] = 774
```

The trace information requested in the simulator command file, trace.regs, is printed.

```
2045: changing d register while dato active
changing ba register while bus busy
2045: r[6] = 772
```

The simulator recognizes that clkoff is missing in the microinstruction at control store location 2044. The clkoff has intentionally be deleted. Note that MICRO/40 did not recognize this error.

```
>e
```

As the discovered error is type 1, the simulation is terminated to correct the error.

```
%  ed fastc.mic
```

The UNIX text editor is called to correct the file, fastc.mic.

```
1846
```

The number of characters in fastc.mic is printed.

```
: ? r4?
```

The microinstruciton, d_r4; dato;, at control store location 2044 is located in the file, fastc.mic.

```
d_r4; dato;
:s/$/clkoff/p
d_r4; dato; clkoff
:w
1853
:q
```

The erroneous microinstruction is corrected.

```
%  mic -s fastc.mic
fastc.mic
91 lines read.
```

The microprogram, fastc.mic, is reassembled.

```
> x set.regs
> x trace.regs
> o 210
> g 2000
```

The simulator is again set up for the invokation of the microporgram with a $210_8$ instruction.

```
2003: r[6] = 776
2037: r[5] = 0
2042: r[5] = 776
2043: r[6] = 774
2045: r[6] = 772
2047: r[6] = 770
2051: r[6] = 776
2053: r[7] = 0
```

The trace information requested in the simulator command file, trace.regs, is printed.

```
16: address out of bounds
```

The microprogram simulation leaves the WCS 11/40 control store address space with the microinstruction, goto 16, at control store location 2054.

```
> x print.regs
```

A listing of the register contents is requested.

```
r1: 10
r2: 12
r3: 14
r5: 776
r6: 766
r7: 0
```

The simulator lists the requested register contents.

```
>_r2 = 0
>_r3 = 0
>_r4 = 0
> o 211
```

The simulator is set up for the vokation of the microprogram with a $211_8$ instruction.

```
> g 2000
```

The simulation is started. Remember the trace requests have not been removed.

```
2013: r[1] = 776
2014: r[1] = 774
2016: r[4] = 16
2017: r[1] = 772
2021: r[3] = 14
2022: r[1] = 770
2024: r[2] = 12
```

```
2026: r[6] = 776
2027: r[5] = 0
2030: r[6] = 1000
```

The trace information is printed.

```
2032: bus command or clkoff ignored
2032: r[6] = 1002
```

The simulator recognizes another error that has not been recognized by MICRO/40.

```
>e
```

The simulation is terminated to correct the error.

```
%  ed fastc.mic
1853
:/rts/;+4p
ba_r6; dati; clkoff  !rts pc
d_r6+2; r6_d; clkoff
r7_unibus; but 16
goto 16
end
: ? rts?
ba_r6; dati; clkoff  ! rts pc
: s/clkoff//p
ba_r6; dati  !rts pc
:w
1847
:q
```

The error is corrected using the UNIX text editor. The error was artificially inserted into fastc.mic

```
%  mic -s fastc.mic
fastc.mic
91 lines read.
```

The corrected microprogram is reassembled.

<fastc.mic is simulated after being invoked by a $210_8$ instruction>

```
>_r2 = 0
>_r3 = 0
>_r4 = 0
>o 211
>g 2000
```

The simulation is reinvoked with a $211_8$ instruction

```
2013: r[1] = 776
      .
      .
      .
2030: r[6] = 1000
2032: r[6] = 1002
2033: r[7] = 0
16: address out of bounds
```

The requested trace information is printed, and the microprogram leaves the WCS 11/40 control store address space with the microinstruction, goto 16, at control store location 2034.

```
> x print.regs
r1: 770
r2: 12
r3: 14
r4: 16
r5: 0
r6: 1002
r7: 0
```

The register contents after the execution of the $211_8$ instruction are printed

```
>i
```

The simulator is reinitialized

```
> x set.regs
> b r6
> o 210
> g 2000
```

A simulation of fastc.mic ($210_8$ instruction) with a break on r6 is started.

```
break 2003: r[6] = 776
>g
break 2043: r[6] = 774
>g
break 2045: r[6] = 772
>g
break 2047: r[6] = 770
>g
break 2051: r[6] = 766
```

```
>g
16: address out of bounds
```

The microprogram leaves the WCS 11/40 control store address space at location 2054.

```
>r
```

All breaks are removed.

```
> o 211
> g 2000
```

A simulation run for a $211_8$ instruction is started.

```
16: address out of bounds
```

The microprogram leaves the WCS 11/40 control store address space at location 2034.

```
>i
> x set.regs
> o 210
> s 2000,30
```

The simulator is reinitialized, and a simulation run for a $210_8$ instruction in stepping mode is started.

```
2000
2001
  :
2053
2054
address out of bounds
```

The microprogram leaves the WCS 11/40 control store address space at location 2054.

```
> b 2017
> o 211
> g 2000
```

A break at control store location 2017 is set for a simulation run with a $211_8$ instruction.

```
break: 2017
> x print.regs
r1: 772
r2: 12
r3: 14
r4: 16
r5: 776
```

```
r6: 766
r7: 0
```

After the break at location 2017, the register contents are printed.

```
>g
16: address out of bounds
```

The microprogram leaves the WCS 11/40 control store address space at location 2034.

```
> = 1000
1000: 0
> = 776
776: 0
> = 774
774: 16
> 772
772: 14
> 770
770: 12
> = 766
766: 0
> = 764
764: 0
```

The contents of the simulated core locations 764 to 1000 are investigated.

```
>e
```

Exit from the simulator.

```
% ed core
core: cannot open
:a
764: 0
0
50
52
54
0
0
.
:w
21
:q
```

A core file (locations 764 to 1000) is created

```
%  sim fastc.bin fastc.tab
```

The simulator is called (not invoked by MICRO/40).

```
> x set.regs
>_r5 = 0776
>_r6 = 766
>0 211
>1 core
> g 2000
```

The simulator is set up for a simulation run with a $211_8$ instruction.

```
16: address out of bounds
> x print.regs
r1: 770
r2: 50
r3: 52
r4: 54
r5: 0
r6: 1002
r7: 0
```

After the execution of the $211_8$ instruction, the register contents are tested.

```
>i
> x set.reg
> o 210
> g 2000
16: address out of bounds
>c
6680.  6680.
```

The time for the execution of a $210_8$ instruction is measured

```
> o 211
> g 2000
16: address out of bounds
>c
14020.  7340.
```

The accumulative execution time for both a $210_8$ and a $211_8$ instruction as well as the time for a $211_8$ instruction are printed.

```
>e
```

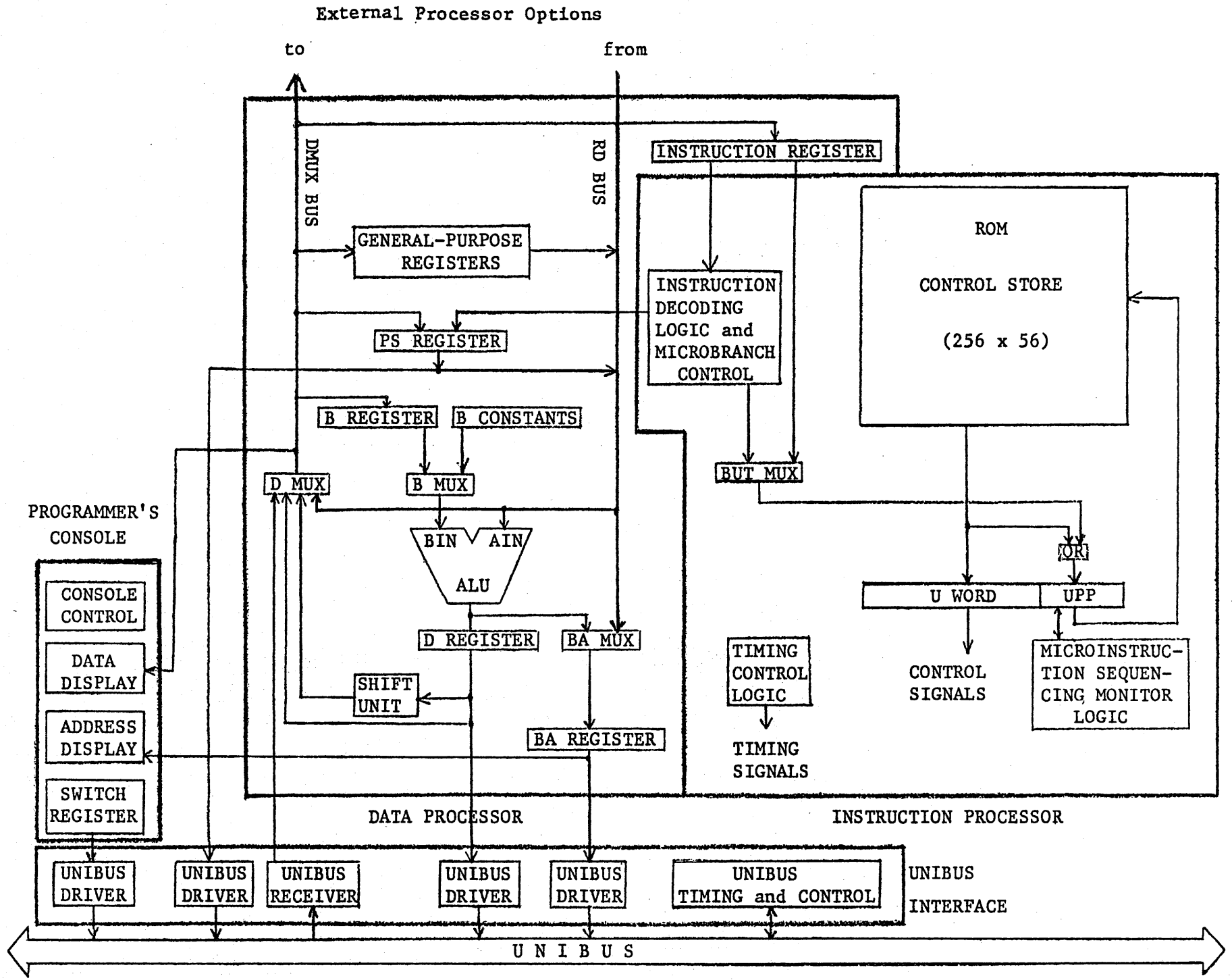
Exit from the simulator.

Acknowledgement

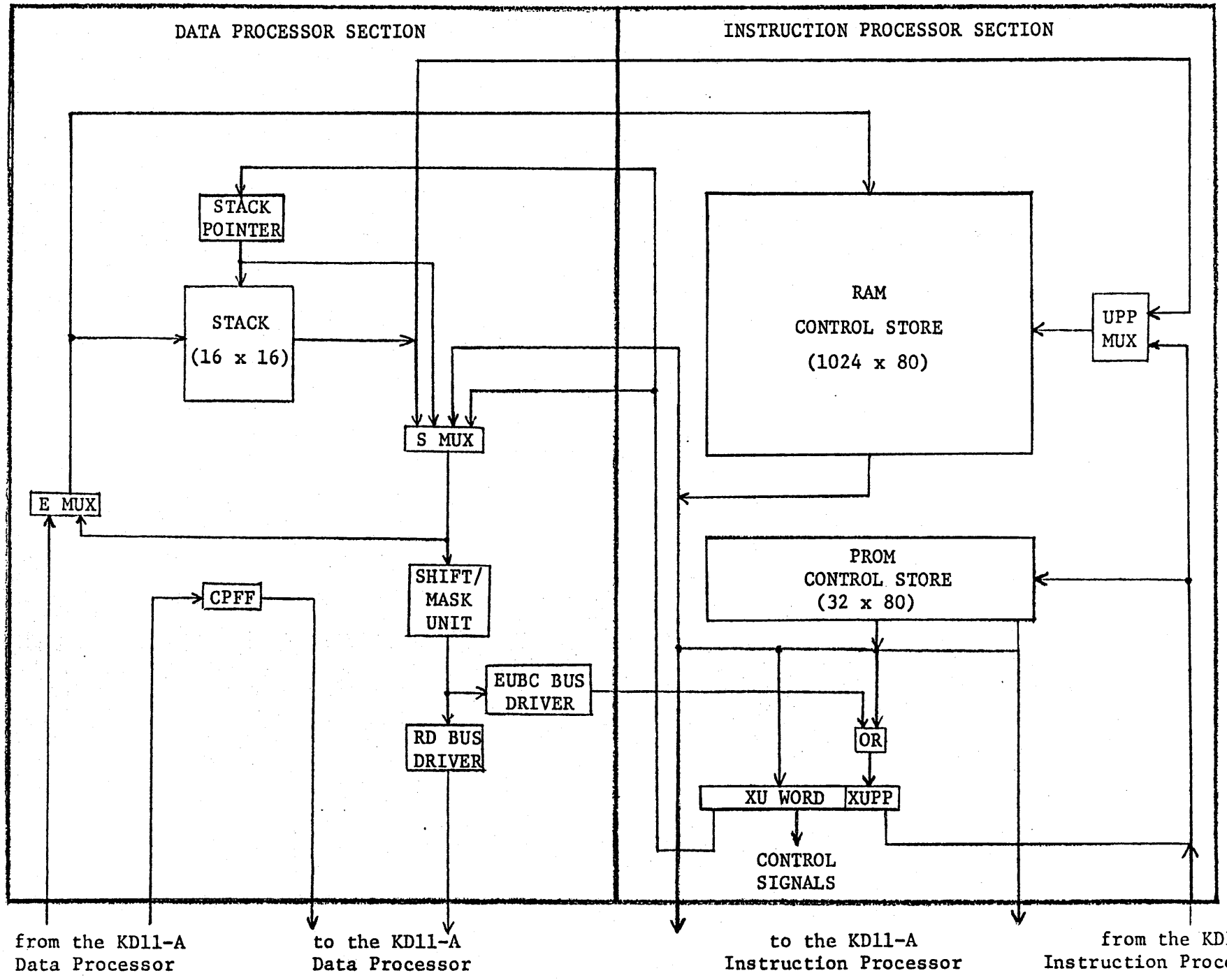
The microcode simulator was originally implemented by R. Kallerhoff of the Technical University Berlin. The authors are indebted to H. Mauersberg, also with the Technical University Berlin, for providing us with the microcode simulator and for many helpful suggestions concerning the development of a microprogramming laboratory around a PDP-11/40E. They also wish to express their gratitude to Professors W. K. Giloi and W. R. Franta for initiating the microprogramming laboratory project at the University of Minnesota. The microprogramming laboratory is funded by University Computer Services, University of Minnesota.

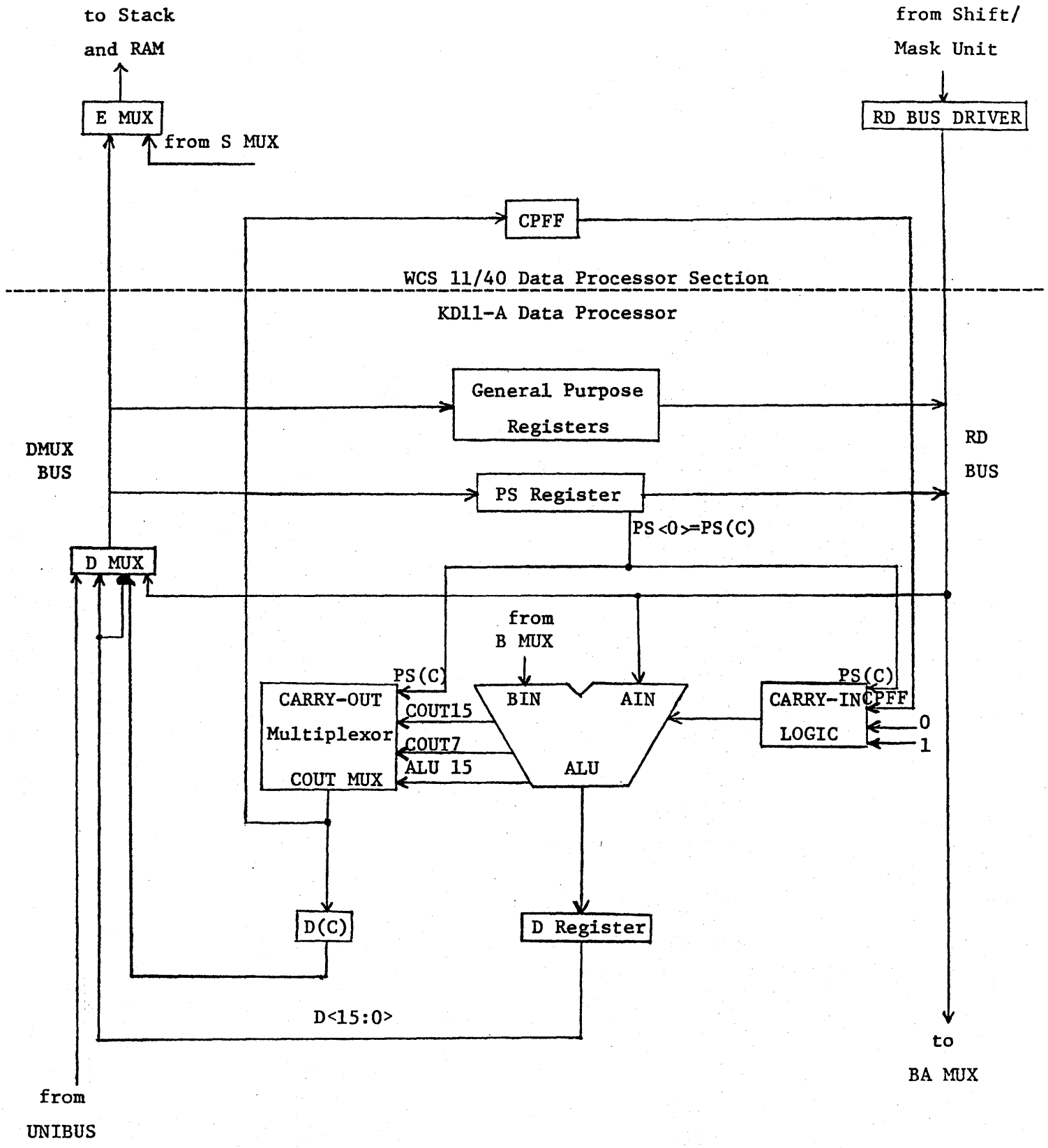
# Appendix

## PDP-11/40E Register-Transfer Block Diagrams

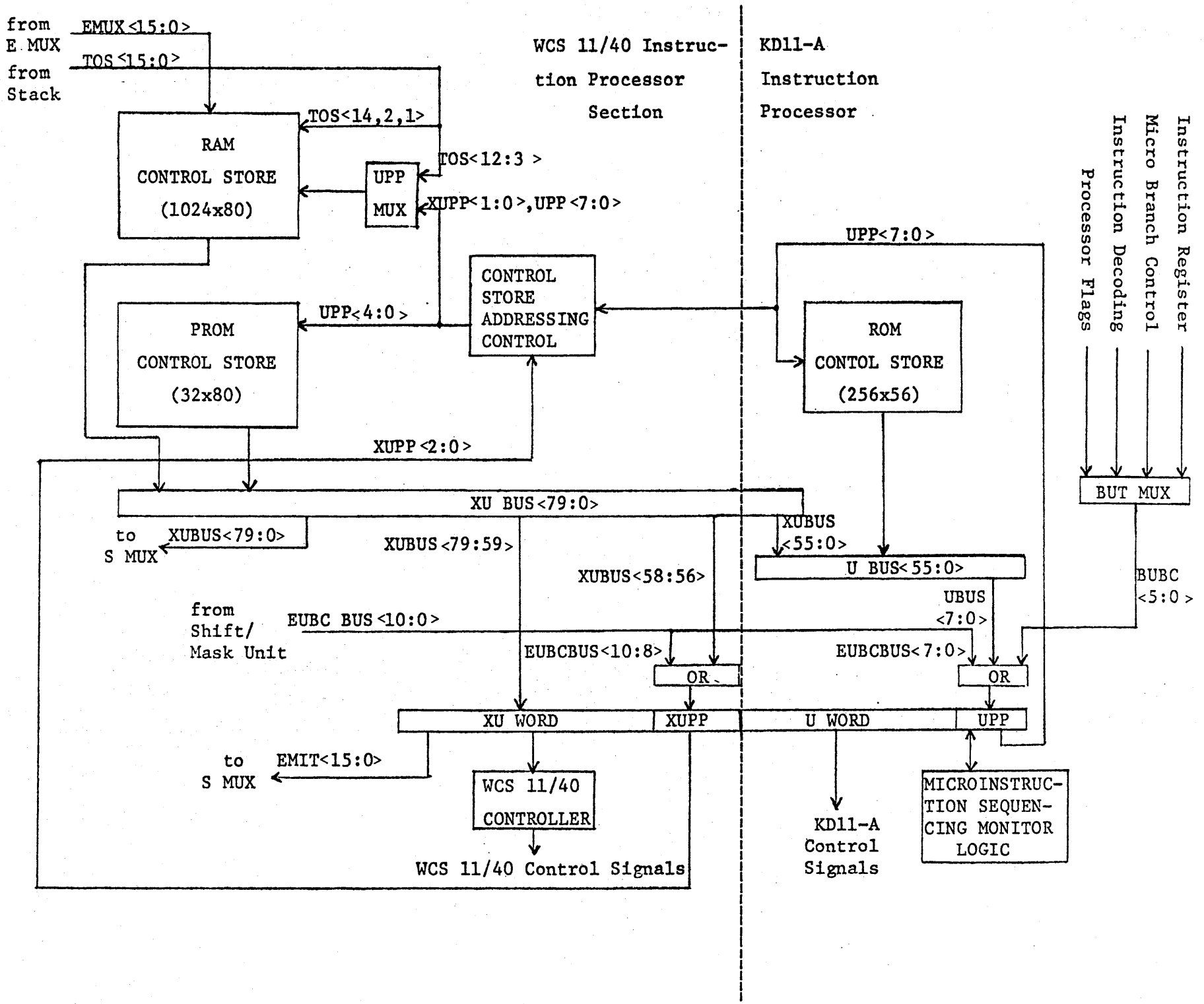



KD11-A Processor Register-Transfer Diagram


DATA PROCESSOR SECTION
INSTRUCTION PROCESSOR SECTION
STACK POINTER
STACK (16 x 16)
S MUX
E MUX
CPFF
SHIFT/ MASK UNIT
EUBC BUS DRIVER
RD BUS DRIVER
RAM CONTROL STORE (1024 x 80)
UPP MUX
PROM CONTROL STORE (32 x 80)
OR
XU WORD
XUPP
CONTROL SIGNALS
from the KD11-A Data Processor
to the KD11-A Data Processor
to the KD11-A Instruction Processor
from the KD11-A Instruction Processor




Data Processor Interface


from E MUX
EMUX<15:0>
from Stack
TOS<15:0>
WCS 11/40 Instruction Processor Section
KD11-A Instruction Processor
TOS<14,2,1>
RAM CONTROL STORE (1024x80)
TOS<12:3>
UPP MUX
XUPP<1:0>,UPP<7:0>
UPP<7:0>
CONTROL STORE ADDRESSING CONTROL
PROM CONTROL STORE (32x80)
UPP<4:0>
ROM CONTOL STORE (256x56)
XUPP<2:0>
Processor Flags
Instruction Decoding
Micro Branch Control
Instruction Register
BUT MUX
XU BUS<79:0>
to S MUX
XUBUS<79:0>
XUBUS<79:59>
XUBUS<58:56>
XUBUS<55:0>
U BUS<55:0>
BUBC<5:0>
UBUS<7:0>
from Shift/ Mask Unit
EUBC BUS<10:0>
EUBCBUS<10:8>
EUBCBUS<7:0>
OR
OR
XU WORD
XUPP
U WORD
UPP
to S MUX
EMIT<15:0>
WCS 11/40 CONTROLLER
WCS 11/40 Control Signals
KD11-A Control Signals
MICROINSTRUCTION SEQUENCING MONITOR LOGIC

## References

[1] Fuller, S. H., Almes, G. T.; Broadley, W. H.; Forgy, C. L.; Karlton,P. L.; Lesser, V. R.; Teter, J. R.,"PDP-11/40E Microprogramming Reference Manual," Department of Computer Science, Carnegie-Mellon University, Tech. Report 16-Jan-76.

[2] UNIX Documentation Book I, "Introduction to UNIX," Computer Systems Laboratory, Department of Computer Science, University of Minnesota.

[3] UNIX Documentation Book II, "UNIX Programmer's Manual Section I - Commands," Computer Systems Laboratory, Department of Computer Science, University of Minnesota.

[4] UNIX Documentation Book III, "The "C" Programming Language," Computer Systems Laboratory, Department of Computer Science, University of Minnesota.

[5] Teter, J. R., "PDP-11/40E Hardware Maintenance Manual," Department of Computer Science, Carnegie-Mellon University, September 1976, revised June 1977.

[6] Berg, H. K., "A PDP-11/40E Microprogramming Primer," Department of Computer Science, University of Minnesota, Tech. Report 78-8.

[7] Berg, H. K.; Dekel, E., "MICRO/40 Assembler Primer," Department of Computer Science, University of Minnesota, Tech. Report 78-9.

[8] Berg, H. K.; Samari Kermani, N., "A Primer on the SMILE Microprogram Load and Test System," Department of Computer Science, University of Minnesota, Tech. Report 78-11.

[9] Berg, H. K.; Covey, C. R., "A Primer on the Use of a Logic State Analyzer as a Microprogram Debugging Aid," Department of Computer Science, University of Minnesota, Tech. Report 78-12.

[10] Mueller, J., "SMILE - Manual," Institut für Softwaretechnik und Theoretische Informatik, Fachgebiet Betriebssysteme, Technical University Berlin, December 1976.