VAX 11/780 CAE TOOLS FOR
MULTIPROCESSOR SIMULATION:
N.mPc User's and Application Manual
and Installation Guide
VAX 11/780 CAE TOOLS FOR
MULTIPROCESSOR SIMULATION:
N.mPc User’s and Application Manual
and Installation Guide

SEPTEMBER 1984

Prepared By: Mr. M. Streit
Approved By: Dr. S.A. Mahmoud
Dr. C. Laferriere

INTELLITECH CANADA LIMITED
352 MacLaren Street
Ottawa, Ontario
K2P 0M6
TITLE: VAX 11/780 CAE TOOLS FOR MULTIPROCESSOR SIMULATION:
N.mPc User's and Application Manual and Installation Guide

AUTHOR(S): Max Streit
INTELLITECH CANADA LIMITED
352 MacLaren Street
Ottawa, Ontario

ISSUED BY CONTRACTOR AS REPORT NO: INT-84-30

PREPARED BY: Max Streit

DEPARTMENT OF SUPPLY AND SERVICES CONTRACT NO: OER 83-05075

DOC SCIENTIFIC AUTHORITY: Michel Savoie
Communications Research Centre
Ottawa, Ontario

CLASSIFICATION: Unclassified

This report presents the views of the author(s). Publication of this report does not constitute DOC approval of the reports findings or conclusions. This report is available outside the department by special arrangement.

DATE: SEPTEMBER 1984
SUMMARY

This manual, a deliverable under contract OER 83-05075 for the Communication Research Centre of the Department of Communications, Government of Canada, describes the N.mPc system and its environment on the Analysis and Simulation laboratory’s VAX 11/780 computer. It will enable a user to interact with the N.mPc system with minimal effort by means of graduated examples which the user can carry out on his/her terminal. This document complements the original set of N.mPc documentation distributed with the system by extending the descriptions to cover the enhancements incorporated to the original N.mPc package under this contract. The examples illustrated in this manual include a simulation of the RCA 1802 microprocessor and a simulation of the Intel 86/12 Single Board computer which features software development with the high-level language C. Other microprocessor simulations used by Intellitech include the Intel 8085, the Texas Instruments SBP 9989 and the Ferranti F-100L. Hints for users of N.mPc’s hardware description language (ISP') and information about the use of all other simulations currently available on the system are given in this document. Installation guidelines for the N.mPc package are also included in this manual.
TABLE OF CONTENTS

1. INTRODUCTION.................................................................1
   1.1. Overview of N.mPc....................................................1
   1.2. N.mPc User Options..................................................3
   1.3. Structure of the Manual.............................................6
   1.4. Associated Documentation.........................................6

2. OVERVIEW OF IMPLEMENTED SIMULATIONS..............................8

3. THE N.mPc DIRECTORY STRUCTURE ON THE VAX........................10

4. N.mPc FILE NAMING CONVENTIONS........................................14

5. HOW TO USE N.mPc COMPONENTS..........................................17
   5.1. Hardware Modeling..................................................17
   5.2. Simulation Building...............................................22
       5.2.1. Introduction................................................22
       5.2.2. Simulation Building Using The Standard Software Path...25
       5.2.3. Simulation Building When Using An Enhanced Software
       Development for 8086 CPU Based N.mPc Simulation..............29
   5.3. Running A Simulation..............................................30

6. SIMULATION EXAMPLES....................................................33
   6.1. Sorting Numbers on a Simulated RCA 1802 Microprocessor......33
   6.2. Running an Assembly Program on a Simulated Intel 86/12 SBC....45
   6.3. Searching For Prime Numbers by Running a C-Program on the
       Simulated Intel 86/12 SBC.........................................57

References................................................................................66

Appendices

A. N.mPc Microprocessor Description Library

B. N.mPc Directory Listings

C. Listings of "PRIME.C", "PRIME.S" Programs

D. N.mPc Installation Guide
**LIST OF FIGURES**

1-1: Elements of the N.mPc System ........................................ 2
1-2: N.mPc User Options ........................................ 5

3-1: The N.mPc Directory Structure on VAX/VMS .......................... 11

5-1: Connection of Ports in ISP ........................................ 18
5-2: Incorrect Implementation of a Jump Instruction .................. 19
5-3: Correct Implementation of a Jump Instruction ........................ 19
5-4: Incorrect Output Procedure .................................. 20
5-5: Correct Output Procedure .................................. 21
5-6: Incorrect CPI Implementation .................................. 21
5-7: Correct CPI Implementation .................................. 22
5-8: Integration of Hardware and Software in N.mPc .................. 24
5-10: Standard Simulation Building in N.mPc ....................... 27
5-11: Enhanced Software Development Environment for 8086 CPU based N.mPc Simulation .................................................. 31

6-1: The Simulated Hardware in the RCA 1802 Example .................. 35
6-2: The RCA 1802 Simulation Example ................................ 36
6-3: Directory Inspection ........................................ 37
6-4: Compiling Hardware Modules .................................. 38
6-5: Standard Software Development .................................. 38
6-6: The User Program in a Simulated Memory .......................... 39
6-7: Integration of Hardware and Software ............................ 40
6-8: Putting a Simulation into the Runtime Mode ....................... 41
6-9: Inspection of a Simulated Memory ................................ 43
6-10: A Simulation Run ........................................ 44
6-11: Inspecting the Simulation Results ................................ 45
6-12: The Hardware Configuration of the Intel 86/12 Simulation .... 46
6-13: Memory Map of the Intel 86/12 Simulation .......................... 47
6-14: Flowchart and target 8086 Assembly Program for squaring numbers .... 49
6-15: Simulation A: An Assembly Program on an Intel 86/12 ........... 50
6-16: Software Development (Cross Assembler Output) ............... 52
6-17: Software Development (Cross Loader and OTOL Output) ....... 53
6-18: The "86ASMOTOL" Command File ............................... 54
6-19: The Software Development Output File .......................... 54
6-20: Putting the User Program into Simulated Memory ................ 55
6-21: Integration of Hardware and Software ............................ 55
6-22: Running an Assembly Program on an 86/12 Simulation ........... 56
6-23: Simulation B: A C Program on an Intel 86/12 .................... 58
6-24: Simulation B: Intel 86/12 Memory Arrangement & Initialization ....... 60
6-25: Cross Compiling a C Program .................................. 61
6-26: Cross Assembling/Linking/Loading a C Program .................. 62
6-27: Integration of the Developed Software .......................... 63
6-28: Running a C Program on a Simulated 86/12 SBC .................. 63
6-29: Results Obtained from Running a C Program on Simulated 86/12 SBC Hardware .................................................. 65
1. INTRODUCTION

1.1. Overview of N.mPc

It is assumed that the reader, having read the "N.mPc Detailed System Description" document [16], is already familiar with the elements of the N.mPc system. A short overview of N.mPc is given here as a reminder.

N.mPc consists of six components which are the metamicro assembler, the linking/loader, the ISP compiler, the ecologist, the simulated memory processor and the run-time environment. These components are used, either singly or in conjunction with others, to specify the behaviour of hardware systems at the register transfer level. If the modeled hardware is programmable, some of these components can be used to provide the capabilities of writing assembly level programs and to produce code which can be executed by the simulated hardware. The N.mPc system also provides for the integration of hardware and software modules so that a simulation can be built and eventually executed on a host computer.

The Meta-micro Assembler and the linking loader are used to generate the software which is to be executed by the simulated hardware. Both are driven by a description of the instruction set of the target machines and can be made to generate code for either vertically or horizontally programmed machines. The linking loader produces code which is executed by a simulated processor or by an actual machine. The ISP compiler is used to produce simulation modules for individual processors and other hardware components of a network. The input language of the compiler is the ISP language which allows the specification of states for the implementation of processor registers and flags, memories for the simulation of memory, and ports which allow
Figure 1-1 Elements of the N.mPc System
input to and output from simulated hardware.

The N.mPc ecologist and a simulated memory processor link the ISP processor modules with the linking loader outputs in order to form complete simulations. A run-time package is used to execute a simulation and to allow extensive user interaction with the simulation.

1.2. N.mPc User Options

There are several options open to a user of the N.mPc system. Figure 1-2 gives an overview of these user options.

Option A1 (in Fig. 1-2): This option represents the standard software development methodology based on the meta micro assembler and the linking/loader. As both of these N.mPc components are programmable, some work is required to produce the necessary metamicro and linking/loader command files. Programming the linking/loader requires a good deal of knowledge about the linking/loader syntax and the addressing modes of the target machine. This option provides, however, the ultimate in flexibility.

Option A2: This option allows the use of cross software development tools if "OTOL", a program that is part of the N.mPc system, can transform the resulting object code into N.mPc's "1.out" object format (see OTOL directory for more information). Option A2 is ideal if a programmable assembler is not required and the
use of a high level language for software
development is desired.

Depending on the complexity of the hardware to
be designed, hardware modeling may be a
demanding task. Option B1 involves the
building of models for the simulated hardware
using ISP™, the hardware specification language
of N.mPc.

Of course there is no need to design hardware
descriptions that may already exist. The "lib"
directory (see section 3) contains some
hardware descriptions designed by Intellitech
and a library of microprocessor descriptions
delivered with the N.mPc package. The hardware
descriptions developed by Intellitech have been
thoroughly tested but the descriptions provided
with the N.mPc system should be checked
carefully before being used.

A listing of the contents of the microprocessor
description library delivered with the N.mPc
package is to be found in Appendix A.

Microprocessor descriptions that were designed
and/or used for simulations by Intellitech are
presented in section 2 of this document.
A) Software Development

- Are there any programmable HW modules?
  - Yes
    - Is a programmable assembler required?
      - Yes
        - Is cross software development tool available?
          - Yes
            - Can OTOL handle target object code format?
              - Yes
                - Change, upgrade module
              - No
                - Write descriptions of programmable assembler, linking loader
        - No
          - Descriptions of programmable assembler linking loader exist in library?
            - Yes
              - A1) Programmable MetaMicro Assembler, Linking Loader
            - No
              - No
    - No
      - B1) Design new HW modules

B) Hardware Modeling

- Required HW modules in library?
  - Yes
    - Check library module
  - No
    - Can library module be used without changes?
      - Yes
        - B2) Use Modules from library
      - No
        - Change, upgrade module

NHW/SW Integration

Simulation

Figure 1-2 N.mPc User Options
1.3. **Structure of the Manual**

This manual is divided into six sections. Section 1 gives a short overview of the elements of the N.mPc system and outlines the options open to the N.mPc user. Section 2 briefly covers the simulations implemented by Intellitech on the present N.mPc installation on the VAX 11-780 of the Analysis and Simulation Laboratory at CRC.

Section 3 shows the directories of the above mentioned N.mPc installation. A short description of the contents of each directory is also given. Section 4 states the standard file naming conventions. Section 5 shows how to use the N.mPc system without referring to a particular example.

Section 6 shows the reader some practical examples which can be executed from a terminal in order to get hands-on experience with N.mPc. The example featuring the RCA 1802 microprocessor uses N.mPc's metamicro assembler for the development of the simulation software whereas the examples on the Intel 86/12 uses a "C" cross-compiler to generate the software and a special program to transform the object code to N.mPc format.

1.4. **Associated Documentation**

Section 1.3 in the System Manual [16] refers to all N.mPc related documents which are also listed in the "References" section of this document. The first document issued in connection with the upcoming release of N.2, the new ISP manual [15], is of remarkable practical value for the user.

Another deliverable under this contract, a report on the validation of N.mPc for microprocessor simulations [20], illustrates the
simulation of a complete Intel 86/12 Single Board Computer and the use of a commercial cross compiler package for the development of the simulation software.
2. OVERVIEW OF IMPLEMENTED SIMULATIONS

An N.mPc simulation requires several components to be in place before it can be built and run. These components include the ISP" hardware descriptions, the simulation software, the topology of the system, etc., and are located in files. These files, in turn, are grouped into directories. For the sake of convenience, all the files which are a logical part of a simulation are assembled into the same directory.

This section will give a brief description of the various simulations implemented by Intellitech. All these "ready-to-run" simulations are located in subdirectories of the "READYSIM" directory (see section 3). Each simulation directory contains a OREADME.FST file with all necessary information. Each simulation is referred to by the name of the subdirectory in which it resides.

86demo: The 86demo directory contains an algorithm searching for prime numbers which is implemented in C and runs on a simulated 8086 based Intel 86/12 SBC. This simulation is described in great detail in section 6.3 of this manual.

86va1: The 86va1 directory contains a simple attitude control algorithm implemented in C and runs on a simulated 8086 based Intel 86/12 SBC. This simulation is described in detail in [20].

86sieve: This directory contains the "SIEVE" benchmark program in C which is to be run on the simulated Intel 86/12 SBC as well as on the actual hardware.

86int: The 86int directory contains a simple program written in assembly language which should be run on the simulated Intel 86/12 SBC. This program is contiuously interrupted.
by interrupts which are handled by a simulated
Programmable Interrupt Controller (PIC). This simulation
shows the interrupt capabilities of the simulated 86/12
SBC.

8085: In this directory a simple assembly language program
decreasing registers can be run on a simulated Intel 8085
CPU.

1802demo: In this directory, one finds an assembly language program
sorting odd and even numbers. The program is
subsequently run on a simulated RCA1802 CPU. This
simulation is described in great detail in section 6.1 of
this document.

SBP9989: The simulation in this directory involves a program
written in assembly language to be run on a simulated
Texas Instruments SBP9989 CPU. This simulation is
described in [8].

F100L: A multiprocessor simulation documented in an Intellitech
technical report [23] uses several simulated Ferranti
F100L CPUs and is to be found in this directory. Five
F100L CPUs perform numerical calculations and coordinate
their activities through a Multibus and a global memory.

multi85: The simulation in this directory is similar to that found
in F100L except that Intel 8085 microprocessors are used.
Further details can be found in [23].
3. THE N.mPc DIRECTORY STRUCTURE ON THE VAX

Figure 3-1 shows the present state of the N.mPc directory structure implemented on the VAX of the Analysis and Simulation Laboratory at CRC. Additional details concerning the directories of the N.mPc package (located in SYS$SYSDISK:[package.nmpc]) are to be found in Appendix D.

The Lantech C 8086 cross software development tool is installed in the "86build" directory. Details about the files implementing the Lantech 8086 C cross software development tool are found in the corresponding Lantech manual [19].

The "OTOL" directory contains source and object code of the "OTOL" program (see Section 2). The "nmec" directory contains a structure of subdirectories that are aimed at facilitating the development of microprocessor simulations using N.mPc.

The "softgen" directory contains two subdirectories which contain the necessary description files if the development of the simulation software is to be done according to the standard way, as represented by option A1. The two subdirectories are listed below:

mmpd: This directory holds the necessary command files for the metamicro assembler. The metamicro is a programmable assembler and the command files are used to enable it to process the assembly language of a given machine.

llcf: This directory contains the necessary command files for the linking/loader. As the metamicro, the linking/loader is programmable and needs to be instructed as to how to resolve the addressing of object code for a particular machine. The command files in llcf are meant to perform this task.
Figure 3-1 The N.mPc Directory Structure on VAX/VMS
The subdirectories of the "lib" directory form a library of hardware descriptions in ISP', the hardware description language of the N.mPc system. They are listed below:

**microlib:** This directory contains the original library of microprocessor description files delivered with the N.mPc system.

**lib8612:** This directory contains the hardware description files (ISP' modules, topology file) necessary to build the simulation of the Intel 86/12 SBC.

**lib8085:** This directory contains the hardware description files necessary to build the simulation of the Intel 8085 CPU.

**lib1802:** This directory contains the hardware description files necessary to build the simulation of the RCA 1802.

**lib9989:** This directory contains the hardware description files necessary to build the simulation of the SBP 9989.

**lib100L:** This directory contains the hardware description files necessary to build the simulation of the Ferranti F100L.

The "86softdev" directory is used to do software development for the simulation of the 8086 based 86/12 SBC (see section 5; option A2 in Figure 1-2). The various subdirectories of 86softdev are listed below:

**c86:** This directory contains the command file that invokes the Lantech C 8086 crosscompiler. All the crosscompiling of C programs is done in c86 which therefore contains a number of C programs and their crosscompiled assembly code versions. These programs can be run on the 86/12 simulation when the software development is completed.

**as86:** Command files invoking the cross assembler and
linker/loader are to be found in this directory. Therefore, the software development stages from assembly code to Intel Hex object code in N.mPc's "1.out" format (see section 5) can be done in as86.

Besides the above mentioned command files, as86 contains assembly code files (.s extension) produced by the Lantech cross compiler and their object versions in N.mPc compatible format (.out extension).

llib: This directory contains the link library associated with the Lantech cross software development tool. Currently, this library contains routines that support doubleword arithmetic on the 8086 CPU.

The "readysim" directory contains "ready-to-run" simulations implemented by Intellitech. The contents of each directory were described in section 2.
4. N.mPc FILE NAMING CONVENTIONS

The file names used in N.mPc simulations are subject to certain conventions. Figure 5-10 shows all the filenames encountered when building an N.mPc simulation the standard way.

The prefixes of the input filenames (iname, tname, pgmname, lname) may be chosen by the user whereas the suffixes (.isp, .t, .m, .i) are compulsory. The other filenames are subsequently created by N.mPc. A short description of each file type used in an N.mPc simulation follows:

pgmname.m
A "m" file contains a user program which will be processed by the metamicro assembler. A successful assembly produces a corresponding "pgmname.n" file.

pgmname.n
This is an intermediate file produced by the metamicro assembler and used by the linking/loader allocator.

lname.i
This file is the linking/loader command file which contains the specification of the address resolution process for a given machine (stored in [icll.nmpc.softgen.11cf]).

iname.a
The output of the linking/loader interpreter. This file is used by the allocator to direct the address resolution process.

1.out
The output of the allocator always uses this same name and if the file is to be saved, it should be renamed. The file 1.out contains a real machine core image, suitable for simulation after processing by the simulated memory processor (SMP).

iname.isp
This file is the input to the ISP' compiler; it contains the ISP' source code describing a simulated piece of hardware.
iname.obj  The output of the ISP^ compiler.

tname.t  The topology file represents the configuration of the hardware to be simulated. The ecologist will build a program called "tname", which will be the executable simulation.

tname.s  A symbol table file created by the ecologist and used by the runtime package. "tname" is the simulation name.

tname.f  The memory list file produced by the ecologist and used by the SMP. It contains the names of all memories used in a simulation.

initmemname;  The name declared for each memory in the "initial" declaration which is part of the topology file. The semicolon should not be forgotten if VMS is to be prevented from adding unwanted suffixes!

initmemname.p  A processed file created by the SMP. It corresponds to a previous linking/loader output file which has been renamed and is used by the simulation program and the simulated memory processor.

pgmname.out  The contents of l.out are overwritten by each creation of a simulation. To share a "ready to run" set of files without going through the whole creation process, the contents of l.out have to be saved into another file (e.g. by doing rename l.out pgmname.out).

pname.x  Another SMP output, containing global symbols from the metamicro; there is one such file per simulation.

tname.d  The simulation data file contains data to be processed by the post processor. This is a new feature of the VAX/VMS version of N.mPc.
The executable simulation program.

The user-programmed description of the (metamicro) assembler for the microprocessor being simulated; it has to be stored in the usual N.mPc directory structure and can be reached by the path [icl1.mmpc.softgen.mmpd].

A few additional filenames are encountered when using the enhanced software development environment for 8086 based N.mPc simulations shown in Figure 5-11:

A program written in the high level language C.

A program written in the 8086 assembly language used by the Lantech cross software development tool. This program may have been written by a programmer or may be the crosscompiled version of a C program.

An object code file in Intel’s HEX format.

An object code file in N.mPc’s "l.out" format.
5. HOW TO USE N.mPc COMPONENTS

5.1. Hardware Modeling

Section 5.1 intends to give some hints to users who are already familiar with N.mPc hardware description language (ISP'). First time users may want to skip this section.

The key to the effective use of N.mPc is to avoid a number of subtle errors when creating hardware descriptions using the ISP' hardware description language. Some ISP' features which are not easily understood at the beginning are summarized in chapter 8 of the "N.2 ISP' User's Manual" [15]; others will be explained here.

Sharing a resource among several devices happens frequently in microprocessor systems. Figure 5-1 shows that one electrical connection may be used to signal whether the resource is in use or not. This means that only one device may have its busy line active while the others have to wait for "busy" to go to a passive state in order to activate their own signal line. ISP' handles this situation in a similar fashion.

Connecting ports together means connecting them to the same "signal". A "signal" represents the common digital value "0" or "1" associated to all ports connected to it. In N.mPc, the common value of several ports connected together is always an "OR" function of all these ports. Two rules therefore have to be strictly respected if more than two ports are connected together in an N.mPc simulation:

1) Only one port at a time may be active otherwise the port values will not correspond to a physical reality.

2) Given N.mPc's "OR" connection function, rule 1 can only be satisfied using positive logic (1 = active, 0 = passive) for all multiple connections.

Another condition for correct hardware modeling is to have a
electrical connections in reality:

Device 1  Device 2  Device 3

modelling of electrical connections by N.mPc:

port 1

port 2 OR busy "signal"

port 3

Figure 5-1 Connection of Ports in ISP
perfect understanding of how the hardware to be modeled should work. An example taken from the ISP description of the Intel 8086 CPU will highlight this issue. A "Jump Short" instruction has erroneously been implemented as shown in Figure 5-2:

```plaintext
0353: (ip = ip + getQ sxt 16;
    fifo_empty = true;
    D(2);
  )!
```

**FIGURE 5-2: Incorrect Implementation of a Jump Instruction**

The important point to remember is that the instruction pointer ("ip") is always increased, so that it can point to the next instruction, before that instruction is executed. This applies to the "JUMP" instruction mentioned above. The actual jump is executed by adding an offset ("getQ sxt 16") to the instruction pointer by using the "getQ" procedure which results in the instruction pointer pointing to the next instruction. In ISP expressions are evaluated from left to right; it therefore becomes clear that the offset is added to the instruction pointer before the latter is increased (by the contents of the "getQ" procedure). The resulting jumps will always be one location too far or too short. The "JUMP" implementations shown in Figure 5-3, has the "getQ" procedure increase the instruction pointer before an offset is added, resulting in a correct jump:

```plaintext
0353: (temp = getQ sxt 16; next;
  ip = ip + temp;
  fifo_empty = true;
  D(2);
  )!
```

**FIGURE 5-3: Correct Implementation of a Jump Instruction**

19
Another important feature of ISP is that assignments are not immediately performed but occur only when the following "next" statement is executed. If this is not kept in mind, errors may occur because assignments may be performed too late or not at all. In the output procedure shown in Figure 5-4, no values are outputted because the assignment "IO = Output;" is never performed due to a missing "next" statement.

```
do_output (addr<WORD>, BW):=
(  EB_offset = addr;
    WRD_BYTE = BW;
    EB_mio = io;
    EB_data = case BW
        word_op: ax
        byte_op: al ext 16
    esac;
    IO = Output;
    while (IO neq DONE) delay (1);
)
```

FIGURE 5-4: Incorrect Output Procedure

When a "next" statement after "IO = Output;" was added, the new output procedure, shown in Figure 5-5, worked correctly:
do_output: requests the BIU for output; waits till done.

```
do_output (addr<WORD>, BW) :=

   EB offset = addr;
   WRD_BYTE = BW;

   EB_mio = io;          !it is device store

   EB_data = case BW
      word_op: ax
      byte_op: al ext 16
   esac;

   IO = Output;next;     !request BIU for Output

   while (IO neq DONE) delay (1); !wait till done.
```

FIGURE 5-5: Correct Output Procedure

Due to the nature of N.mPc and the VMS operating system, certain errors in the ISP code will cause a VMS instigated abort of the simulation run. Faced with such a situation, it is evident that an ISP programmer will have to resort to less than ideal debugging methods.

The following comments and examples are of interest for the time being.

```
07: (  
    tdata = getdata (getEA, not wrd3);
    temp = alu (tdata, getimd_sw, Sub, not wrd3);
    IFREG D (4); IFNOTREG D (17);
  ) !CPI
```

FIGURE 5-6: Incorrect CPI Implementation

In the implementation shown in Figure 5-6, the CPI instruction is supposed to compare either two word arguments or two byte arguments. The instruction bit carrying the word/byte information is addressed by
the macro "wrdf". The above "CPI" implementation, however, tests the wrong bit ("wrdf") which has nothing to do with the format of this instruction. Contradictions between the instruction format and the actual size of the operands may therefore occur and will cause VMS to abort the simulation catastrophically. The corrected "CPI" instruction, shown in Figure 5-7, tests the bit that really indicates word/byte format (represented by the macro "wrdf") and works as intended. This is the kind of error to look for when facing the situation of a simulation dropping out of the runtime mode because of an error in a hardware module.

```
07: (*
    temp2=getEA;next;
    temp = getdata (temp2, wrdf);
    next;
    temp = alu(temp, getimd_sw, Sub, wrdf);
    next;
    IFREG D(4); IFNOTREG D(17);
) !CPI
```

FIGURE 5-7: Correct CPI Implementation

Finally, learners of the ISP language should be reminded that the N.2 ISP User's Manual [15] is the manual they should be using as the N.2 system will eventually supersede N.mPc entirely.

5.2. Simulation Building

5.2.1. Introduction

From the reading of the "N.mPc Detailed System Description" [16] and section 1.1 of this document, the reader should be familiar with the elements of the N.mPc system. Section 5.2.1 explains in a few words how the N.mPc elements are used to build a simulation. Section 5.2.2 then gives a general example for building a simulation. Finally section 5.2.3
describes the same simulation building process in the case of the enhanced software development environment for 8086 based N.mPc simulations. In the latter section, only the different commands for software development are explained as all other steps are exactly the same as in the example of section 5.2.2.

Figure 5-8 shows the main activities involved when N.mPc elements are used to build a simulation. A "Hardware Description Language" called ISP is used to model microprocessors, memories, support chips, buses, etc., at the register transfer level. Two different methods exist to bring the target software to be run on the modeled hardware into the simulation. The standard software path uses a "metamicro" assembler which has to be programmed for every type of microprocessor. The metamicro assembler is not always entirely compatible with commercial assemblers and produces object code in the 1.out format used by N.mPc. An alternative method of developing the simulation software makes use of commercial cross compilers and cross assemblers which are available for most microprocessors. The source code for the simulation software, written in a high level language or directly in a commercial assembler, is transformed into object code which would not be compatible with the "1.out" format of N.mPc. A program called "OTOL" (object to 1.out) transforms the object code to the 1.out format (see [20] for details).

Once the desired hardware and software descriptions are entered into the N.mPc system, everything can be integrated to create a "Simulation Runtime Environment". This runtime environment allows the execution of the developed software on the modeled hardware; it also supports various observation, debugging and data collection facilities. The following sections will describe the use of N.mPc's hardware and software paths in detail.
Figure 5-8  Integration of Hardware and Software in N.mPc
5.2.2. Simulation Building Using The Standard Software Path

To use the standard software path mentioned in section 5.2.1, the description files for software generation have to reside in a given directory structure which is depicted in Figure 5-9. This structure is mandatory as it is required by the assembling linking-loading functions of N.mPc (the "mas" command). The hardware description files and the user programs are usually put into one directory whose actual location is left to the user. Figure 3-1 shows the complete directory structure for the use of N.mPc on the Analysis and Simulations laboratory's VAX 11-780 computer. The conventions mentioned in section 4 have been respected and the simulation descriptions for different hardware configurations are grouped into one directory per configuration. "Logical names", a VMS feature, were used to reach each directory directly, thus avoiding the need to go through the VMS tree directory structure.

Figure 5-10 shows the hardware and software description inputs, the major components of the N.mPc system and the sequence of commands necessary to "build" a simulation. In other words, Figure 5-10 shows all the complete simulation building process all the way to the "ready-to-run" state. The sequence of the commands used to build a simulation will now be explained.

Simulation building means giving a sequence of commands to components of the N.mPc system. Those commands are described below, in the order in which they would normally be given:

1) Compile hardware modules (command: "ic iname.isp"):

This command invokes the compiler for the ISP* hardware description language. The ISP* compiler is invoked once for each hardware module and supports various options (see [17, 15]).
NMPC.DIR

SOFTGEN.DIR

ICLI.DIR

MMPD.DIR

(Neural Network Processor) (Metamicro Assembler Descriptions)

Figure 5-9 Directory Structure Required for Standard Software Development
Figure 5-10 Standard Simulation Building in N.mPc
2) **Standard software development (command: "mas-I[pathname] iname pgmname.")**

"mas" is an N.mPc command which performs the sequential invocation of the metamicro assembler and of the linking loader. It produces core image files in the .1.out format. The following points are worth mentioning with respect to the structure of the "mas" command.

i) "pathname" indicates the path to the linking/loader description; on VAX/VMS it is for instance "[icll.nmpc.softgen.11cf]".

ii) "iname" is the name (without suffix!) of the metamicro assembler description.

iii) "pgmname.m" is the name of the user program to be run.

iv) Details about the use of "mas" can be found at the end of [3] and in [18].

3) **Saving object code (command: "rename 1.out pgmname.out"),
Distributing object code to simulated memory (command: "copy pgmname.out initmemname.;"):**

The first command prevents the transformed, loadable user code from being overwritten by every subsequent simulation. To get a "out" file loaded into a certain simulated memory, a copy of it with a name corresponding to the name declared in this memory's "initial" declaration has to be created. This is how the SMP knows to which memory a certain initial memory content (eg. user program) is to be transferred.

4) **Integration of hardware and software (command: "ec tname"):**

This command causes the system to prompt the user with the message: "hardware, software or both." Typing "h", "s" or "b" in response to the prompt instructs "ec" to integrate changed or new hardware, software or both in order to form a "ready-to-run" simulation. "ec" invokes the ecologist (hardware integration) and the SMP
software integration). The ecologist is described in detail in [4] and [18]. The ecologist and the SMP can also be invoked independently by typing a "h" or "s" in answer to "ec's" prompt. In particular, it is advisable to run the SMP before any simulation is rerun to avoid leaving spurious values produced by earlier runs in the simulated memories.

5.2.3. Simulation Building When Using An Enhanced Software Development For 8086 CPU Based N.mPc Simulation

1) "@cc8086 name": This command executes a VMS command file that invokes all of the C compiler passes (see [19,20] for details), producing an 8086 assembly output file with a ".s" suffix. If the target software is to be written in assembly, this command is of course not used.

2) "@86asmotol name": This command invokes a command file which goes through the steps of assembling, linking, loading and transferring to the "l.out" format of an assembly source program. This assembly source program may have been generated by the C cross compiler or written independently. Details are also found in [19, 20]. Automatic renaming of 1.out to name.out is also performed.

3) "copy name.out initmemname.;": This command (step 3 in Figure 5-11) is only mentioned to show that it is equivalent to step 3 in Figure 5-10. The renaming of the 1.out file produced by the OTOL program is automatically done in the corresponding command file. The above command for distributing object code to simulated memories is identical to the one used in the standard software development path.

The simulation building process when using the enhanced software
development is precisely the same as the one described in the previous section. The difference is that the software development step does not involve any programmable metamicro assembler and linking loader. Instead, a commercial cross software development tool (in this case a C 8086 cross software tool by Lantech Inc.) and the "OTOL" program (part of the N.mPc system) are used to produce object code in N.mPc's "1.out" format. Programs in high level language (C in this case) as well as in assembly can be handled by the enhanced software development environment to produce code for a certain target hardware (here the 8086 CPU). Detailed information about the enhanced software development for 8086 based simulations is found in [19], [20].

When building a simulation using the enhanced software development environment, step 2 (software development) of Figure 5-10 is replaced by the following commands, which are further highlighted in Figure 5-11:

5.3. Running A Simulation

Once a simulation has been built, it is put into the "runtime mode" by the "run" command; the run command corresponds to step 5 of Figure 5-10. The exact command is: "run tname" where tname stands for the name of the topology file in use. A new set of commands is available in the runtime mode in order to run a simulation effectively.

The use of N.mPc's runtime facilities is described in [1,11,12] but is most easily learnt by using the "help" facility available in the runtime mode (command: "help help"). Consequently, just a few comments to help the user with the data collection and debugging functions are made here.
Figure 5-11 Enhanced Software Development Environment for 8086 CPU based N.mPc Simulation
It is useful to explain how one refers to certain variables or ports in hardware modules. A "processor name" has to be declared in the topology file ("processor" declaration) for each hardware module which is part of the simulation. Each hardware module also uses a certain number of internal variables called states and ports. To refer to a certain state or port on a given hardware module, one uses the processor name followed by a colon and the name of the state or port. For example, cpu2:PC refers to the variable or port PC of the processor cpu2. If no processor name is mentioned before the colon, the processor name of the first module to be declared in the topology file will be assumed.

The following steps allow the user to display lines of ISP\(^\text{e}\) code on the terminal while they are being executed. This facility is a very powerful debugging feature.

1) Compile "name.isp" using the -t option (ic -t name.isp) or "T_on;/T_off;" statements to display all or part of the ISP\(^\text{e}\) code (see also [15,17,21]).

2) Put a copy of the ISP\(^\text{e}\) source file in question into the RMS stream format using the FCVT utility. The command is "fcvt name.isp name.ill".

3) Once in the runtime mode, the "ispline" command activates the line display facility: "ispline processor name on name.ill" (the "processor name" is declared for each module in the topology file).

During a simulation, it is possible to collect relevant data and store them in a file. The "trace" command is used in the runtime mode to collect data about states, ports, etc.. After running the simulation, the collected data can be inspected using the "pp" command (see [22]).
6. SIMULATION EXAMPLES

The examples in this section show how to use the N.mPc system for building and running N.mPc simulations. Example 6.1 shows the standard software development method for an RCA 1802 based microprocessor simulation. Example 6.2 and 6.3 both involve a simulation of the Intel 86/12 SBC using the enhanced software development environment for 8086 based N.mPc simulations. In example 6.2, a program written in assembly language is run on the 86/12 8086 CPU whereas a program written in a high level language (C) is run in example 6.3. It is suggested that these examples be repeated by the user on his/her own terminal.

6.1. Sorting Numbers on a Simulated RCA 1802 Microprocessor

This section includes a listing from an interactive N.mPc session in which a simulation involving an RCA 1802 microprocessor was built and run. The listing is complemented with comments where additional explanations are required.

The simulation needs five description files in order to be built and run:

- Descriptions of the hardware configuration ("cos.t": topology file), the hardware module ("r1802a.isp") and a user program written in metamicro assembler ("tl.m") are in a common simulation directory ("1802demo").

- Descriptions of the programmable "metamicro assembler" ("r1802.m") and the linking loader ("r1802.l") are in their special directories called "mmpd" and "llcf".
The hardware modeled in this example consists of an RCA 1802 microprocessor with internal memory and is shown in Figure 6-1. The user program is located at the bottom of the memory, in locations 0 to 44. It sorts integer numbers into odd and even numbers. Initially, the unsorted numbers are stored in locations 45 to 247; the odd number will be placed in locations 300 to 393 while the even numbers will be in 400 to 505. A flag q is set at the end of operations so that a breakpoint can be set to stop the simulation.

The necessary steps for building the simulation will now be explained in detail. The listing of the sequence of commands has to be broken into several smaller parts which are enclosed in a box to make them stand out from the text. Figure 6-3 shows a diagram of this simulation and the sequence of commands to build it. An inspection of the relevant directories for this particular simulation shows that all the 5 description files for the RCA 1802 simulation are present as shown in Figure 6-2:
Figure 6-1 The Simulated Hardware in the RCA 1802 Example
Figure 6-2 The RCA 1802 Simulation Example
If the "1802demo" directory listings show more files than what is listed above, it indicates that intermediate files have not been cleaned up following a previous simulation building. The command "@ reset" should therefore be used to initiate the cleaning process. The steps to build the simulation are listed below:

1) The first step to build a complete simulation is compiling the hardware description file(s) as shown in Figure 6-4:
$ ic r1802a.isp
Pass 1 complete
Warnings: 0
Errors: 0
Assembling

$ dir
Directory USER$DISK1:[ICL1.NMPC.READYSIM.1802DEMO]

FIGURE 6-4: Compiling Hardware Modules

It can be seen from Figure 6-4 that the compilation of the source hardware description(s) produces an object file in .obj format, if the compiler does not encounter fatal errors.

2) The second step invokes the "standard" software path; is shown in Figure 6-5:

$ mas -I[icl1.nmpc.softgen.11cf] r1802 tl.m
MAS VAX/VMS version 1.0
executing <$micro tl.m>
executing <$cater [icl1.nmpc.softgen.11cf]r1802a tl.n>

$ dir
Directory USER$DISK1:[ICL1.NMPC.READYSIM.1802DEMO]

FIGURE 6-5: Standard Software Development
In Figure 6-5 "[icll.nmpc.softgen.llcf]" is the directory path to the linking loader description file (r1802.i). The "mas" program does the assembling ("micro"), linking ("inter") and loading ("cater") of a user program (tl.m), thereby producing an "l.out" output file. The "l.out" file contains a core image ready to be loaded into the processor's memory. The metamicro assembler also produces an intermediate output file in the ".n" format; it is a non executable object file.

3) The third step in this simulation building example, selects the simulated memory in which the user program is to be stored. Figure 6-6 shows what happens on the user's terminal.

```
$ rename l.out tl.out
$ copy tl.out coreimage.;
$ dir

Directory USER$DISK1:[ICLL.NMPC.READYSIM.1802DEMO]

@README.FST;1       COREIMAGE.;1       COS.T;1       DEMO.TXT;1
R1802A.ISP;13       R1802A.OBJ;1      RESET.COM;5       TL.M;6
TL.N;1             TL.0UT;1

Total of 10 files.
```

FIGURE 6-6: The User Program in a Simulated Memory

The command "rename l.out tl.out" in Figure 6-6 prevents the processed object version of the user program from being overwritten by the next simulation output.

The "copy tl.out coreimage.;" command is given for the following reason. A simulation may contain several simulated memories. The SMP takes the user programs in the "l.out" object format and loads them into one of the simulated memories as an initial content. The
user tells the SMP in which simulated memory to put the object code representing his program. This is done by creating a copy of his program with a name corresponding to the name declared for each simulated memory in the topology file ("Initial" declaration). At the same time, the use of the "copy" command (rather than "rename") preserves a copy of the processed user program which can later be used to reload the user program in case of simulation changes or to load the user program into other simulated memories.

(4) The fourth step in this simulation building exercise integrates the hardware and software components of the simulation. Step 4 is depicted in Figure 6-7.

$ ec cos
Change hardware, software, or both? (h or s or b): b
executing <$ecologist cos>
Assembling ...
Linking ...
executing <$smp cos>

$ dir
Directory USER$DISK1:[ICL.NMPC.READYSIM.1802DEMO]

<table>
<thead>
<tr>
<th>ØREADME.FST;1</th>
<th>COREIMAGE.;1</th>
<th>COREIMAGE.P;1</th>
<th>COS.EXE;1</th>
</tr>
</thead>
<tbody>
<tr>
<td>COS.F;1</td>
<td>COS.S;1</td>
<td>COS.T;1</td>
<td>COS.X;1</td>
</tr>
<tr>
<td>DEMO.TXT;1</td>
<td>R1802A.ISP;13</td>
<td>R1802A.OBJ;1</td>
<td>RESET.COM;5</td>
</tr>
<tr>
<td>T1.M;6</td>
<td>T1.N;1</td>
<td>T1.OUT;1</td>
<td></td>
</tr>
</tbody>
</table>

Total of 15 files.

FIGURE 6-7: Integration of Hardware and Software

Figure 6-7 shows that several things happened. Answering the prompt of the "ec" command with a "b" for both (hardware and software) gets the hardware ("ecologist") and the software ("smp") integrated to form an executable simulation. This is done by
running the ecologist and the simulated memory processor against the topology file. Several files are created by the "ecologist":

- cos.s: a symbol list
- cos.f: a memory list
- cos.exe: the simulation execution program

The SMP creates two files:

- cos.x: the memory symbol list
- coreimage.p: the page-formatted executable user code

The "ecologist" and the "smp" can be invoked separately by choosing the appropriate answer to the prompt after invoking "ec" ("s" for a software change and "h" for a hardware change).

5) The simulation is now ready to be put into the runtime mode.

```
$ run cos
Welcome to N.mPc/VMS
N.mPc: cos
#
```

**FIGURE 6-8: Putting a Simulation into the Runtime Mode**

Figure 6-8 shows that running the simulation program (.exe format) puts the simulation into the "runtime mode". The user is greeted by a system message and the runtime environment displays a # as a prompt. The command "q" allows to leave the runtime mode. A "help" facility explains all the runtime commands. The execution of the "reset.com" command file puts the simulation back into the initial state; it would be ready for another learning session. The command is "@reset".
The next part of this section will show how to run the "tl.m" user program in the RCA 1802 simulation. To put into perspectives what the next few figures will show, some runtime commands (see [11]) are explained here.

- The "states" command shows the internal variables used by the hardware descriptions which are part of the simulation.
- The "ports" command shows the ports used for external communication; in this simulation there are no ports used.
- The "memory :m 0 99" command will display the selected area of memory.

Some comments regarding the contents of the memory of the simulated processor will also be made.

- The memory locations between the first and the second decimal numbers are displayed.
- The locations 0 to 44 contain the user code.
- The locations 45 to 247 contain the unsorted numbers.
- The locations 300 to 393 contain zeroes initially and the odd numbers afterwards (check!).
- The locations 400 to 505 contain the even sorted numbers after the simulation (check!).

To check whether the user program really sorts the numbers, one should check the unsorted numbers and the resulting numbers before and after execution of the program. The memory inspection shown in Figure 6-9 makes sure that the numbers to be sorted are odd and even and that no numbers (other than 0) are in the memory locations reserved for the sorted numbers before the execution of the sorting program.
FIGURE 6-9: Inspection of a Simulated Memory
Breakpoints are used to stop a simulation during or at the end of a user program's execution. Figure 6-10 shows how the simulation is run and stopped by a breakpoint.

```
# bkpt :q change
breakpoint number 1
#
# disp :q change
monitor number 2
#
# run
t: 4040

2 disp :q change = 1
simulation halted by bkpt 1
(bkpt :q change)
```

FIGURE 6-10: A Simulation Run

In connection with the running of the simulation, several points should be stressed:

- The "bkpt:q change" command sets a breakpoint which will eventually stop the simulation when the q flag changes.
- Breakpoints can also be set on various conditions related to states and ports (see runtime "help" facility).
- The "run" command starts execution of the simulation software which can only be stopped by:
  - a breakpoint,
  - a control/C,
  - fatal errors in the program.

The correctness of the program can be checked by inspecting the relevant memory areas after the execution of the program. Figure 6-11 shows how to inspect the simulation results.
FIGURE 6-11: Inspecting the Simulation Results

Obviously, the sorting in odd and even numbers was successful. The "q" command is used to exit from the runtime mode. The "@reset" command cleans the directory for the next user of this example.

6.2. Running an Assembly Program on a Simulated Intel 86/12 SBC

The hardware used in this example is the Intel 86/12 single board computer. The hardware description in ISP" for the 86/12 was developed by Intellitech and is described in detail in reference [20]. The hardware and memory configurations of the 86/12 under test are shown in Figures 6-12 and 6-13 respectively. The memory of the 86/12 is, in fact, made up of three separate but contiguous memory modules: read-only, read/write and global memory modules.
Figure 6-12 The Hardware Configuration of the Intel 86/12 Simulation
### RAM (accessed via Multibus, 1k)

<table>
<thead>
<tr>
<th>(16383)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM (accessed via Multibus, 1k)</td>
</tr>
</tbody>
</table>

### Global Memory (1k):

<table>
<thead>
<tr>
<th>(15360)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Memory (1k):</td>
</tr>
<tr>
<td>initial memory name: &quot;gblcore:;&quot;</td>
</tr>
<tr>
<td>memory reference name: &quot;gmem:me&quot;</td>
</tr>
</tbody>
</table>

### ROM (13 k):

<table>
<thead>
<tr>
<th>(15359)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROM (13 k):</td>
</tr>
<tr>
<td>initial memory name: &quot;romcore:;&quot;</td>
</tr>
<tr>
<td>memory reference name: &quot;rom:mem86&quot;</td>
</tr>
</tbody>
</table>

### RAM (1k):

<table>
<thead>
<tr>
<th>(14336)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM (1k):</td>
</tr>
<tr>
<td>initial memory name: &quot;ramcore:;&quot;</td>
</tr>
<tr>
<td>RAM reference name: &quot;ram:me&quot;</td>
</tr>
</tbody>
</table>

### Figure 6-13 Memory Map of the Intel 86/12 Simulation
The simulation building procedure is very much the same in this case as it was for the previous example involving the RCA 1802. However, since the objective is to build an 8086 CPU based N.mPc simulation, the enhanced software environment, described in Section 5.3, is used to produce the simulation software.

The target program of this example uses the processing unit of the 86/12 configuration: the 8086 CPU. The target program continuously squares the numbers from 1 to 100. As the program is in an infinite loop, the "Control C" command is used to stop the simulation. Figure 6-14 shows a flowchart as well as a listing of the squaring program; the program name is "86ass.s".

The hardware for this example is to be found in the directory "86demo". Figure 6-15 shows the steps necessary to run an Intel assembly program on 8086-based hardware. These steps are further explained below:

1) "@86asmotol 86ass":

This command invokes the Lantech cross assembler/linker/loader using a command file. Details about particular commands in the command file can be found in the relevant Lantech Manual [19] and in [20]. The assembly source file "86ass.s" (".s" suffix is expected by the command file!) is transformed into the file "86ass.out" which contains the N.mPc compatible object code. A ".hex" intermediate output file in the Intel HEX format is also produced.

2) "copy 86ass.out romcore.;":

If not otherwise chosen through the "-c" loader option, the code produced by the Lantech cross assembler is loaded starting at memory location 1024. As this is identical with the starting address of the
Figure 6-14  Flowchart and target 8086 Assembly Program for squaring numbers
Figure 6-15 Simulation A: An Assembly Program on an Intel 86/12
"ROM" memory area of the 86/12 simulation (see Figure 6-13), the program "86ass.s" is loaded into the "ROM". The ROM has the initial memory name "romcore;".

3) "smp val":

This command invokes the SMP directly. The command "ec" followed by an "s" in answer to the "ec"-prompt has the same effect. The SMP loads the processed user program into the appropriate memory, already chosen by command 2.

4) "run val":

This command puts the simulation into the runtime mode. Before the assembly program can be run, the program counter and the code, data and stack segment registers have to be initialized appropriately. The program counter should point to the beginning of the program which is at memory location 1024 in this case, and the top of the stack should be located near the upper end of the memory area reserved for this program by the loader (memory location 1200 in this example). To observe the activity of the simulation a "display :ax change" command shows the numbers before and after the squaring operation.

The following figures show listings pertaining to the actual execution of the commands discussed above and have been complemented with explanatory notes. However, not all of the intermediate outputs are shown. As for the example, it is to be found in a directory called "86demo" (also see Figure 6-15). Figure 6-16 shows how the Lantech cross software development tool is invoked and the resulting output of the cross assembler.
$ set def 86demo
$ dir 86ass.*

Directory USER$DISK1:[ICL1.NMPC.READYSIM.86DEMO]

86ASS.S;3
Total of 1 file
$
$
$ @86asmotol 86ass

ass8086: 86ass.s

868086 Cross-Assembler Version 5.0 (C) 1982
Source: '86ass.s'
Object: 'tmpl.tmp'

[1][2][3][Gen]

Line Addr S Generated

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28

$000000003

---------------------
* main program *
*squares numbers 1..100 in*
* an infinite loop *
***************************
* Max Streit, Sept. 84 *
***************************

00000003

prog: .segment code

start: mov ax,#0000h
        mov bx,#0000h
        mov cx,#0064h
        here: inc bx
        call square
        loop here

00000003

prog: .ends

000000003

prog: .end

FIGURE 6-16: Software Development (Cross Assembler Output)
Input:  "tmp2.tmp"
Output: "tmp3.tmp"

<table>
<thead>
<tr>
<th>Address</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODE block:</td>
<td>00400 00020</td>
</tr>
<tr>
<td>DATA block:</td>
<td>00430 00000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Segment</th>
<th>Block</th>
<th>Start address</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>prog</td>
<td>C</td>
<td>00400</td>
<td>0016</td>
</tr>
<tr>
<td>$common</td>
<td>D</td>
<td>00430</td>
<td>0000</td>
</tr>
</tbody>
</table>

:0400000300400000B9
:020000020040BC
:1600000B800000B0000B9640043E80400E2FAEBEF8BC3F7E3C38A
:020000020043B9
:00000001FF
DEBUG OPTION --
Input file is -86ass.hex output file is -1.out
MACHINE is i
Initializing the outputfile - 1.out
No Symbol Table
Initializing the inputfile - 86ass.hex
Base Starting Address 1024
  Current Segment = 1024
A data record starting address = 1024 ending address = 1045
  Current Segment = 1072
Writing the data records
  INIT RECORD
writing the init records

FIGURE 6-17: Software Development (Cross Loader and OTOL Output)

It is now necessary to have a closer look at what the "@86asmotol 86ass" command does, its execution results in the steps shown in Figures 6-16 and 6-17. The listing of the "86asmotol.com" command file, depicted in Figure 6-18, shows that it invokes the assembler, the linker, the loader and the "OTOL" program and then performs a renaming of the output to a name with the ".out" suffix.

53
$ asm86 -l -0 tmpl.tmp 'pl'.s
$ lkr86 -l -0 tmp2.tmp tmpl.tmp llib: lib8086.a
$ ldr86 -l -0 tmp3.tmp tmp2.tmp
$ rename tmp3.tmp 'pl'.hex
$ otol -id tmp3.tmp 'pl'.out -a%-4000$0
$ del/nolog *.tmp;

FIGURE 6-18: The "86ASMOTOL" Command File

According to Figure 6-18, an "l.out" format version (".out" suffix) of the "86ass.s" program should have been created. The "dir" command in Figure 6-19 is used to ensure that these files are present.

$ dir 86ass.*
Directory USER$DISK1:[ICL1.NMPC.READYSIM.86DEMO]
86ASS.OUT;1 86ASS.S;3

Total of 3 files.
$
$

FIGURE 6-19: The Software Development Output File

As shown in Figure 6-20, step 2 of this simulation building example gets the "86ass" program ready to be loaded into the simulated ROM (see also section 5.2.2, comments on step 3).
FIGURE 6-20: Putting the User Program into Simulated Memory

The hardware of the 86/12 single board computer having already been integrated in this simulation example, it is sufficient to call the SMP to take care of the "86ass" program. Figure 6-21 shows what happens on the user's terminal.

FIGURE 6-21: Integration of Hardware and Software

The fourth step puts the simulation into the runtime mode. The instruction pointer and stack pointer are then initialized. The "display" runtime command is invoked to monitor the ax register in which the numbers from 1 to 100 and their squares will be observed. The "run" command starts the execution of the "86ass" program, "Control C" stops it and "q" makes the simulation exit from the runtime mode. Figure 6-22 shows the simulation run in detail.
$ run val

welcome to N.mPc/VMS

N.mPc: val
# dep 1024 :ip
# dep 1200 :sp
# disp :ax change
monitor number 1
# run

t: 16500
  1 disp :ax change = 0

t: 68000
  1 disp :ax change = 1

t: 70750
  1 disp :ax change = 1

t: 164000
  1 disp :ax change = 2

t: 166750
  1 disp :ax change = 4

t: 260000
  1 disp :ax change = 3

t: 262750
  1 disp :ax change = 9

t: 356000
  1 disp :ax change = 4

t: 358750
  1 disp :ax change = 16

: : :

t: 1124000
  1 disp :ax change = 12

t: 1126750
  1 disp :ax change = 144
^C
simulation stopped by interrupt
# q

FIGURE 6-22: Running an Assembly Program on an 86/12 Simulation

The user is encouraged to experiment with the different runtime commands while going through this example. An occasional "purge" command will get rid of unnecessary versions of the various files.
6.3. Searching For Prime Numbers by Running a C-Program on the Simulated Intel 86/12 SBC

In this example the target software is written in the "C" high level language and is cross compiled to Intel compatible assembly code. Programs for the 86/12 developed directly in C should be kept to a moderate size. An example will explain what is meant by "moderate". The largest high level program run on an Intel 86/12 simulation so far ("VAL-CMD.C"; see [20]) consists of about 30 lines of "C". These lines are run 80 times in a loop, and perform numerical calculations. The total size of the code is close to 2K bytes and it takes almost 5 hours of CPU time to execute the program on the simulated 86/12 hardware simulation on a VAX 11-780. The reasons for this apparent program size restriction are twofold: Firstly, the emulated hardware runs at a much slower speed than its real counterpart; secondly, the situation will be exacerbated by the inefficiencies introduced by the compiler. In addition to those speed concerns, only programs with integer arithmetic will work properly. In this case the lack of an 8087 numeric co-processor coupled with the lack of an 8087 runtime emulation package are responsible for the limitation to integer arithmetic.

The program run in this example, "prime.c" (see Appendix C), determines the prime numbers in the range of 1 to 15. Figure 6-23 shows how a C-program is run on an 86/12 hardware simulation. The only new step, compared to the previous example given in section 6.2.1, is that the C crosscompiler is now used to produce a program in Intel assembly code ("prime.s", Appendix C). The command file "cc8086.com" invokes all the components of the Lantech C-crosscompiler.
Figure 6-23 Simulation B: A C Program on an Intel 86/12
A memory map showing the location of the user program is depicted in Figure 6-24. The code starts at the beginning of the reserved area in memory (1024); immediately following the code area is a memory block reserved for the data and stack area. This arrangement is important in order to perform a correct initialization of relevant registers of the 8086 CPU. The three initialization steps are listed below:

1) The command "deposit 1024 :ip" will load the program counter with the starting address of the code.

2) The commands "deposit 81 :ss" and "deposit 81 :ds" ensure that the data and stack segments start at the beginning of the common data and stack area chosen by the loader.

3) The command "deposit 204 :sp" puts the top of the stack near the end of the data area reserved by the loader because the stack grows downwards.

The following figures, representing listings from an interactive simulation session, show the commands necessary to run the "prime.c" program on a 86/12 simulated hardware as shown in Figure 6-23. Listings of the programs run in "86demo" are contained in Appendix C.
End of loader-reserved memory area:

User Choice For Top of Stack:

Starting address of common data and stack area:

Start of loader-reserved memory area; start of first code segment:

(inaccessible memory)

End of loader-reserved memory area:

User Choice For Top of Stack:

Starting address of common data and stack area:

Start of loader-reserved memory area; start of first code segment:

(inaccessible memory)

Figure 6-24 Simulation B: Intel 86/12 Memory Arrangement & Initialization

1) "deposit 1024 : ip"

2) "deposit 81 : ss"

3) "deposit 204 : sp"
Figure 6-25 shows the cross compilation of the "prime.c" program.

```bash
$dir prime.*
Directory USER$DISK1:[ICL1.NMPC.READYSIM.86DEMO]
PRIME.C;2
Total of 1 file
$ @cc8086 prime
$ dir prime.*
Directory USER$DISK1:[ICL1.NMPC.READYSIM.86DEMO]
PRIME.C;2 PRIME.S;1
Total of 2 files.
$
```

FIGURE 6-25: Cross Compiling a C Program

The invocation of the cross assembler/linker/loader is illustrated in Figure 6-26 (note: starting addresses of "Code Block" and "Data Block" are shown in the cross loader listing). The whole output listing of the cross assembler is not shown here as it is too long.
$ @86asmotol prime
as8086: prime.s

Lantech Systems, Inc.
8086 Cross-Linker Version 5.0 (C) 1982

Files linked: tmpl.tmp
[sys$library:lib8086.a]:environ.o csv.o cret.o

Lantech Systems, Inc.
8086 Cross-Loader Version 5.0 (C) 1981

Input: "tmp2.tmp"
Output: "tmp3.tmp"

<table>
<thead>
<tr>
<th>Address</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODE block:</td>
<td>00400 00100</td>
</tr>
<tr>
<td>DATA block:</td>
<td>00510 00020</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Segment</th>
<th>Block</th>
<th>Start address</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>C</td>
<td>00400</td>
<td>000D8</td>
</tr>
<tr>
<td>c$csv</td>
<td>C</td>
<td>004E0</td>
<td>000A</td>
</tr>
<tr>
<td>C$cret</td>
<td>C</td>
<td>004F0</td>
<td>000B</td>
</tr>
<tr>
<td>$common</td>
<td>D</td>
<td>00510</td>
<td>0016</td>
</tr>
</tbody>
</table>

$ dir prime.*

Directory USER$DISK1:[ICL1.NMPC.LIB.TESTS.86DEMO]

PRIME.C;3 PRIME.HEX;1 PRIME.OUT;1 PRIME.S;1

Total of 4 files.

FIGURE 6-26: Cross Assembling/Linking/Loading a C Program
Software development of the C program "prime" is now complete. The object version of "prime" ("prime.out") now has to be loaded in the desired simulated memory. Figure 6-27 shows the necessary command to do this as well as the command initiating the integration of the new software into the simulation (by invoking the SMP).

```
$ copy prime.out romcore.;
$ ec val
|Change hardware, software, or both? (h or s or b): s
|executing <$smp val>
|
```

FIGURE 6-27: Integration of the Developed Software

The simulation is now ready to run the C program prime.c on the simulated 86/12 SBC hardware. Figure 6-28 shows how to put the simulation into the runtime environment, initialize the necessary 8086 registers and set a breakpoint that stops the simulation after execution of the program.

```
$ run val
|Welcome to N.mPc/VMS\n|N.mPc: val
|# dep 1024 :ip
|# dep 81 :ss
|# dep 81 :ds
|# dep 204 :sp
|# bkpt :ir eql Oxcb00
|breakpoint number 1
|# run
|simulation halted by bkpt 1
|(bkpt :ir eql Oxcb00)
```

FIGURE 6-28: Running a C Program on a Simulated 86/12 SBC
The results and all the variables used in the program are stored on the stack. A look at the assembly version (see Appendix A, "prime.s") of the program reveals the manner in which the variables are kept. The program "prime.c" will find the prime numbers in the 1 to 15 range. Stack space has been reserved for each number in the range of 1 to 15; the reserved area starts at 1456 and ends at 1485 if the initialization is carried out as indicated before. The prime numbers in the range of 1 to 15 were found to be 1, 2, 3, 5, 7, 11, and 13. The total execution time is about 2 minutes on a VAX-11/780 computer. The resulting prime numbers are represented as an array of flags stored on the stack. There is one flag per number in the chosen range. If a flag is set to 1 after the program execution, the corresponding number is a prime number. Figure 6-29 shows the results found by memory inspection along with an explanatory table.
FIGURE 6-29: Results Obtained from Running a C Program on Simulated 86/12 SBC Hardware

<table>
<thead>
<tr>
<th>Flags</th>
<th>Memory</th>
<th>Prime Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1456)</td>
<td>1/0/1</td>
<td></td>
</tr>
<tr>
<td>(1457)</td>
<td>0/0/0</td>
<td></td>
</tr>
<tr>
<td>(1458)</td>
<td>1/2/2</td>
<td></td>
</tr>
<tr>
<td>(1459)</td>
<td>0/0/0</td>
<td></td>
</tr>
<tr>
<td>(1460)</td>
<td>1/3/3</td>
<td></td>
</tr>
<tr>
<td>(1461)</td>
<td>0/0/0</td>
<td></td>
</tr>
<tr>
<td>(1462)</td>
<td>0/0/0</td>
<td></td>
</tr>
<tr>
<td>(1463)</td>
<td>0/0/0</td>
<td></td>
</tr>
<tr>
<td>(1464)</td>
<td>0/0/0</td>
<td></td>
</tr>
<tr>
<td>(1465)</td>
<td>0/0/0</td>
<td></td>
</tr>
<tr>
<td>(1466)</td>
<td>0/0/0</td>
<td></td>
</tr>
<tr>
<td>(1467)</td>
<td>0/0/0</td>
<td></td>
</tr>
<tr>
<td>(1468)</td>
<td>0/0/0</td>
<td></td>
</tr>
<tr>
<td>(1469)</td>
<td>0/0/0</td>
<td></td>
</tr>
<tr>
<td>(1470)</td>
<td>0/0/0</td>
<td></td>
</tr>
<tr>
<td>(1471)</td>
<td>0/0/0</td>
<td></td>
</tr>
<tr>
<td>(1472)</td>
<td>0/0/0</td>
<td></td>
</tr>
<tr>
<td>(1473)</td>
<td>0/0/0</td>
<td></td>
</tr>
<tr>
<td>(1474)</td>
<td>0/0/0</td>
<td></td>
</tr>
<tr>
<td>(1475)</td>
<td>0/0/0</td>
<td></td>
</tr>
<tr>
<td>(1476)</td>
<td>0/0/0</td>
<td></td>
</tr>
<tr>
<td>(1477)</td>
<td>0/0/0</td>
<td></td>
</tr>
<tr>
<td>(1478)</td>
<td>0/0/0</td>
<td></td>
</tr>
<tr>
<td>(1479)</td>
<td>0/0/0</td>
<td></td>
</tr>
<tr>
<td>(1480)</td>
<td>0/0/0</td>
<td></td>
</tr>
<tr>
<td>(1481)</td>
<td>0/0/0</td>
<td></td>
</tr>
<tr>
<td>(1482)</td>
<td>0/0/0</td>
<td></td>
</tr>
<tr>
<td>(1483)</td>
<td>0/0/0</td>
<td></td>
</tr>
<tr>
<td>(1484)</td>
<td>0/0/0</td>
<td></td>
</tr>
<tr>
<td>(1485)</td>
<td>0/0/0</td>
<td></td>
</tr>
</tbody>
</table>
REFERENCES


REFERENCES CONTINUED


APPENDIX A

N.mPc Microprocessor Description Library
cd microlib

$ dir3

Directory USER$DISK1:[ICL1.NMPC.LIB.MICROLIB]

A2900.DOC:1
A2909A.ISP:1
BWALD1.A:1
BWALD2.A:1
I8080.DIR:1
I8080E.S:1
I8085.M:1
I8086.A:4
I8086A.ISP:1
I8086D.ISP:1
M6502.A:1
M6500.A:1
M68000.A:1
M68000BM.ISP:1
PDP11.I:1
R1802.DIR:1
R1802A.ISP:1
S2650.A:1
S2650A.ISP:1
T9900.I:1
T9900B.ISP:1
TEST.S:1
TRY.L:1
TRY.Q:1
TRY2.M:4
Z80.A:1
Z8000.A:1
Z8000.M:1
Z8000BM.ISP:1
Z80C.ISP:4
A2901A.ISP:1
A2910A.ISP:1
BWALD1.I:1
BWALD2.I:1
I8080E.F:1
I8085.A:2
I8086A.ISP:1
I8086B.ISP:1
ISPOUT.SIM:2
M6502.I:1
M6700.I:1
M68000.DIR:1
M68000A.ISP:1
M68000BM.ISP:1
M6802.M:1
M68000.M:1
M68000.DIR:1
M68000A.ISP:1
M68000BM.ISP:1
M68000.M:1
M68000A.ISP:1
PDP11.M:1
PDP11.M:1
PDP11.M:1
R18022.I:1
R1802C.ISP:1
S2650.I:1
T9900.A:1
T9900.I:1
TEMP.W:2
TEST.T:1
TRY.M:7
TRY1.M:3
TRY2.N:7
VAX11A.ISP:1
Z80.DIR:1
Z80000.DIR:1
Z8000A.ISP:1
Z8000B.ISP:1
Z80A.ISP:1
Z80D.ISP:1
Z80B.ISP:1
Z80E.ISP:1

Total of 96 files.

$
APPENDIX B

N.mPc Directory Listings
cd c86
$ dir3

Directory USER$DISK1:ICL1.NMPC.B6SOFTDEV.C86]

86SIEVE.C;2  86SIEVE.S;1  CC8086.COM;1
CLONGCMD.C;2  CLONGCMD.EXE;1  CLONGCMD.OBJ;1
CLVALID.C;2  CLVALID.EXE;1  CLVALID.OBJ;1
CMD.C;1  CMD.EXE;1  CMD.OBJ;1
CMD.S;1  CP1SEND.C;3  CP1SEND.S;2
CP1TASK.C;4  CP1TASK.S;1  CP2RECV.C;3
CP2RECV.C;1  CP2TASK.C;3  CP2TASK.S;1
CP3RECV.C;1  CP3RECV.S;1  CP3TASK.C;3
CP3TASK.S;1  CP4RECV.C;1  CP4RECV.S;1
CP4TASK.C;3  CP4TASK.S;1  CVALID.C;1
CVALID.EXE;2  CVALID.OBJ;2  EDTINI.EDT;1
FORSACL.DAT;7  FORSACL.EXE;5  FORSACL.FOR;2
FORSACL.OBJ;4  INIT.TXT;2  IVALID.C;2
IOVALID.S;1  LONGCMD.C;4  LONGCMD.S;4
LONGVAL.C;2  LONGVAL.S;1  NMPCCMD.C;8
NMPCCMD.S;7  NMPCCHM.C;2  NMPCCHM.S;1
NMPCPRIME.C;1  NMPCPRIME.OBJ;1  NMPCPRIME.S;1
NMPCSIEVE.C;4  NMPCSIEVE.EXE;2  NMPCSIEVE.OBJ;2
NMPCSIEVE.S;2  PRIME.C;2  PRIME.EXE;1
PRIME.OBJ;1  PRIME.S;1  RECV21.C;12
RECVBYTE.C;4  RECVBYTE.S;2
RECV21.S;11  SACL.EXE;12  SACL.OBJ;12
SEND12.C;10  SEND12.S;7  SENDBYTE.C;2
SENDBYTE.S;1  SIEVE.C;1  SIEVE.EXE;2
SIEVE.OBJ;1  SIMSIEVE.C;2  SIMSIEVE.S;1
VALCHM.C;2  VALCHM.S;1  VALID.C;3
VALID.S;3  VAXSIEVE.C;3  VAXSIEVE.EXE;3
VAXSIEVE.OBJ;3

Total of 82 files.
$
cd as86
$ dir3

Directory USER$DISK1:[ICL1, NMPC, 86SOFTDEV, AS86]

OREADME.FST;2  AS8086.TXT;3
COASMOTOL.COM;4  AS8ASMOTOL.COM;8
EXINTIO.S;21  EDTINI.EDT;1
PRIME.OUT;4  LONGCMD.OUT;3
RETEST.S;4  RECV1.OUT;1
SENDBYTE.OUT;3  RECV21.OUT;3
TERMTEST.S;7  SEND12.S;1
VALCMD.OUT;4  TEST.86;27

Total of 26 files.
$
cd llib
$ dir3

Directory USER$DISK1:\ICLI\NMPC.8&SPTDEV.LLIB

ADD.0;2 ADD.3 ADD.0;2 ADD.3
AND.0;3 CMPL.0;2 CMPL.0;3
CRET.0;2 CRET.0;3 CSV.0;2
CSV.0;3 DIV.0;7 DIV.0;3
EDITINI.EDIT;1 ENVIRON.0;2 ENVIRON.0;2
FSTART.0;2 GETBYTE.0;2 GETBYTE.0;2
IBMSTART.0;2 IN.0;2 IN.0;2
INSTALL.COM;2 INSTALL.LIB;1 LIB8086.A;37
LSHI.0;2 LSHI.0;2 LSHI.0;2
LSHL.0;2 MOD.0;2 MOD.0;2
MODU.0;2 MODU.0;2 MUL.0;2
MUL.0;2 NEG.0;2 NEG.0;2
OR.0;2 OR.0;2 PRINT.0;2
PRINT.0;2 QEMPT21.0;6 QEMPT21.0;6 QEMPT21.0;6
QEMPTY.JOU;1 QEMPTY.0;2 QEMPTY.0;2
QFULL.0;2 QFULL.0;2 QFULL.0;2
QFULL12.0;2 QFULL12.0;2 QFULL12.0;2
RSHI.0;2 RSHI.0;2 RSHI.0;2
RSHL.0;2 START.0;2 START.0;2
SUB.0;2 SUB.0;2 SUB.0;2
SWITCH.0;2 TEST.0;2 TEST.0;2
UDIV.0;2 UDIV.0;2 UDIV.0;2
UMUL.0;2 XMIT12.0;5 XMIT12.0;5
XMITBYTE.0;2 XMITBYTE.0;2 XMITBYTE.0;2
XOR.0;2 XOR.0;2 XOR.0;2

Total of 73 files.
$
set def 86val
$ dir3

Directory USER\DISK1\ICL1.NMPC.READYSIM.86VAL]

OREADME.FST;9
CMD.MAI;2
DPRAM.ISP;11
GBLCORE.;4
GLOBALMEM.OBJ;7
HEXCMD.OBJ;1
INTERRUPT.OBJ;7
MAX86MEM.ISP;36
MULTINT86.OBJ;21
PRINT.S;3
ROMCORE.;77
SACLZ.EXE;1
SACLZI.EXE;2
TERMINAL.OBJ;5
VAL.F;46
VAL.X;184
VALCMD.S;1

CMD.C;2
CMD.MP1;2
DPRAM.OBJ;6
GBLCORE.F;219
HEXCMD.C;2
IN.S;3
MAX86CPU.ISP;5
MAX86MEM.OBJ;44
PIC.ISP;24
RAMCORE.;98
ROMCORE.P;185
SACLZ.FOR;2
SACLZI.FOR;3
VAL.D;257
VAL.S;46
VALCMD.C;2
CMD.EXE;2
CMD.OBJ;1
EDITINI.EDT;1
GLOBALMEM.ISP;10
HEXCMD.EXE;1
INTERRUPT.ISP;7
MAX86CPU.OBJ;5
MULTINT86.ISP;27
PIC.OBJ;20
RAMCORE.P;607
SACLZ.DAT;3
SACLZI.DAT;5
TERMINAL.ISP;9
VAL.EXE;44
VAL.T;5
VALCMD.OUT;3

Total of 49 files.
$
```plaintext
$ cd 86demo
$ dir3

Directory USER$DISK1:[ICL1.NMPC.READYSIM.86DEMO]

<table>
<thead>
<tr>
<th>Filename</th>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>README.FST</td>
<td>F</td>
<td>5</td>
</tr>
<tr>
<td>86ASS.S</td>
<td>S</td>
<td>3</td>
</tr>
<tr>
<td>DPRAM.OBJ</td>
<td>O</td>
<td>6</td>
</tr>
<tr>
<td>GLOBALMEM.ISP</td>
<td>ISP</td>
<td>10</td>
</tr>
<tr>
<td>INTERRUPT.OBJ</td>
<td>O</td>
<td>5</td>
</tr>
<tr>
<td>MAX86CPU.OBJ</td>
<td>O</td>
<td>183</td>
</tr>
<tr>
<td>MULTINT86.ISP</td>
<td>ISP</td>
<td>27</td>
</tr>
<tr>
<td>PIC.OBJ</td>
<td>O</td>
<td>19</td>
</tr>
<tr>
<td>PRIME.OUT</td>
<td>O</td>
<td>3</td>
</tr>
<tr>
<td>ROMCORE.P</td>
<td>P</td>
<td>20</td>
</tr>
<tr>
<td>TERMINAL.OBJ</td>
<td>O</td>
<td>4</td>
</tr>
<tr>
<td>VAL.S</td>
<td>S</td>
<td>18</td>
</tr>
</tbody>
</table>

Total of 35 files.

$ cd 86sieve
$ dir3

Directory USER$DISK1:[ICL1.NMPC.READYSIM.86SIEVE]

<table>
<thead>
<tr>
<th>Filename</th>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>README.FST</td>
<td>F</td>
<td>4</td>
</tr>
<tr>
<td>NMPSIEVE.C</td>
<td>C</td>
<td>6</td>
</tr>
<tr>
<td>RAMCORE.P</td>
<td>P</td>
<td>150</td>
</tr>
<tr>
<td>ROMCORE.P</td>
<td>P</td>
<td>37</td>
</tr>
<tr>
<td>SIEVE.OBJ</td>
<td>O</td>
<td>12</td>
</tr>
<tr>
<td>VAL.F</td>
<td>F</td>
<td>12</td>
</tr>
<tr>
<td>VAL.X</td>
<td>X</td>
<td>72</td>
</tr>
<tr>
<td>VAXSIEVE.OBJ</td>
<td>O</td>
<td>1</td>
</tr>
</tbody>
</table>

Total of 25 files.

$ cd 86int
$ dir3

Directory USER$DISK1:[ICL1.NMPC.READYSIM.86INT]

<table>
<thead>
<tr>
<th>Filename</th>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>README.FST</td>
<td>F</td>
<td>2</td>
</tr>
<tr>
<td>GBLCORE.P</td>
<td>P</td>
<td>82</td>
</tr>
<tr>
<td>MAX86CPU.DAT</td>
<td>DAT</td>
<td>1</td>
</tr>
<tr>
<td>RAMCORE.P</td>
<td>P</td>
<td>91</td>
</tr>
<tr>
<td>VAL.D</td>
<td>D</td>
<td>143</td>
</tr>
<tr>
<td>VAL.S</td>
<td>S</td>
<td>37</td>
</tr>
</tbody>
</table>

Total of 18 files.
```
```

cd 8085
$ dir3

Directory USER$DISK1:[ICL1.NMPC.READYSIM.8085]

OREADME.FST;1   I8085A.ISP;1   I8085A.OBJ;1
IB5TEST.D;16    I85TEST.EXE;3   I85TEST.F;4
IB5TEST.S;4     I85TEST.T;2    I85TEST.X;9
MAXCORE.;8      MAXCORE.P;9    MEM.ISP;1
MEM.OBJ;1       T85.M;6       T85.N;9

Total of 15 files.
$ cd 1802demo
$ dir3

Directory USER$DISK1:[ICL1.NMPC.READYSIM.1802DEMO]

OREADME.FST;1   COREIMAGE.;1   COREIMAGE.P;1
COS.EXE;1       COS.F;1       COS.S;1
COS.T;1         COS.X;1       DEMO.TXT;1
R1802A.ISP;13   R1802A.OBJ;1  RESET.COM;5
T1.M;6

Total of 14 files.
$ cd sbp9989
$ dir3

Directory USER$DISK1:[ICL1.NMPC.READYSIM.SBP9980]

OREADME.FST;2   MAXCORE.;1   MAXCORE.P;2
T9989.ISP;1     T9989.OBJ;1
TEST.D;6        TEST.EXE;1
TEST.M;3        TEST.N;5
TEST.T;2        TEST.X;2
TIMEM.OBJ;1

Total of 16 files.
```
cd lib9989
$ dir3

Directory USER$DISK1:[ICL1.NMPC.LIB.LIR9989]

READ.ME;1   T9989.A;1   T9989.I;1
T9989.ISP;1   T9989.M;1   TEST.M;1
TEST.M;1   TEST.T;1   TIMEM.ISP;1

Total of 9 files.
$
cd lib8612
$ dir3

Directory USER$DISK1:[ICL1,NMPC,LIB,LIB8612]

OREADME,FST;2
GLOBALMEM.ISP;10
INTERRUPT,OBJ;7
MAX86CPU.OBJ;184
MULTINT86.ISP;27
PIC,OBJ;20

Total of 18 files.
$ cd lib8085
$ dir3

Directory USER$DISK1:[ICL1,NMPC,LIB,LIB8085]

18085,A;2
18085,ISP;1
18085,I;1

Total of 6 files.
$ cd lib1802
$ dir3

Directory USER$DISK1:[ICL1,NMPC,LIB,LIB1802]

A,ISP;2
COREIMAGE.;1
COS,D;21
COS,F;21
COS,X;16
COSC,F;16
COSC,X;7
ISPOUT,SIM;12
ISPOUT,SIM;9
ISPOUT,SIM;6
ISPOUT,SIM;3
ISPOUT,SIM;0
L.OUT;17
R1802A,OBJ;7
R1802C,OBJ;12
R1802H,ISP;3
T1,H;2
T2,OBJ;3
Z,F;4
Z,X;1

Total of 58 files.
$ cd
cd microlib
$ dir3

Directory USER$DISK1:[ICL1.NMPC.LIB.MICROLIB]

A2900.BOC;1  A2901A.ISP;1  A2902A.ISP;1
A2909A.ISP;1  A2911A.ISP;1
BWALD1.A;1  BWALD1.I;1  BWALD1.M;1
BWALD2.A;1  BWALD2.I;1  BWALD2.M;2
I8080.DIR;1  I8080.E;F;1  I8080.E.ISP;1
I8080.E.S;1  I8085.A;2  I8085.I;1  I8085.M;1
I8085.M;1  I8085.A.ISP;1  I8085.B.ISP;1
I8085.M;1  I8085.E;F;1  I8085.B.ISP;1
I8086.A;4  I8086.A.ISP;1  I8086.B.ISP;1  I8086.C.ISP;1
I8086.B.ISP;1  ISP.OUT;S;1  L.OUT;1
M6502.A;1  M6502.I;1  M6502.M;1
M6800.A;1  M6800.I;1  M6800.M;1
M68000.A;1  M68000.DIR;1  M68000.I;1  M68000.M;1
M68000.M;1  M68000A.ISP;1  M68000B.ISP;1
M68000BM.ISP;1  NEWFILE.LOG;1  PDP11.A;1
PDP11.I;1  PDP11.M;1  R1802.A;1
R1802.DIR;1  R1802.I;1  R1802.M;1
R1802A.ISP;1  R1802C.ISP;1  R1802CM.ISP;1
S2650.A;1  S2650.I;1  S2650.M;1
S2650A.ISP;1  T9900.A;1
T9900.I;1  T9900.M;1
T9900B.ISP;1  TEMP.W;2  TEST.F;1
TEST.S;1  TEST.T;1  TRIAL.CR;2
TRY.L;1  TRY.M;7  TRY.N;2
TRY.Q;1  TRY1.M;3  TRY1.N;3
TRY2.M;4  TRY2.N;7  VAX11A.ISP;1
Z80.A;1  Z80.BOC;1  Z80.DOC;1
Z80.I;1  Z80.M;1  Z80.M;3
Z8000.A;1  Z8000.DIR;1  Z8000.I;1
Z8000.M;1  Z8000A.ISP;1  Z8000B.ISP;1
Z8000BM.ISP;1  Z80A.ISP;1  Z80B.ISP;1
Z80C.ISP;4  Z80D.ISP;1  Z80E.ISP;1

Total of 96 files.
$
cd mmpd
$ dir3

Directory USER$DISK1:[ICL1.NMPC.SOFTGEN,MMPD]

I8086.M;2  MY86.M;2  MY86.M;1
R1802.M;1  T9989.M;1

Total of 8 files.
$ cd llcf
$ dir3

Directory USER$DISK1:[ICL1.NMPC.SOFTGEN,LLCF]

I8085.A;2  I8085.I;1  I8086.A;6
I8086.I;1  I80861.A;3   I80861.A;2
I80861.I;5  I80861.I;4   R1802.A;1
R1802.I;1  T9989.A;2   T9989.I;1

Total of 12 files.
cd softgen
$ dir3

Directory USER$DISK1:[ICL1,NMPC,SOFTGEN]

LLCF.DIR;1   MMPD.DIR;1

Total of 2 files.
$ cd lib
$ dir3

Directory USER$DISK1:[ICL1,NMPC,LIB]

LIB1802.DIR;1    LIB8085.DIR;1    LIB8612.DIR;1
LIB9989.DIR;1    MICROLIB.DIR;1

Total of 5 files.
$ cd 86softdev
$ dir3

Directory USER$DISK1:[ICL1,NMPC,86SOFTDEV]

AS86.DIR;1    CG86.DIR;1    LLIB.DIR;1

Total of 3 files.
$ cd readysim
$ dir3

Directory USER$DISK1:[ICL1,NMPC,READYSIM]

1802DEMO.DIR;1   8085.DIR;1      86DEMO.DIR;1
86INT.DIR;1      86SIEVE.DIR;1   86VAL.DIR;1
F100L.DIR;1      MULTI85.DIR;1   S8P9989.DIR;1

Total of 9 files.
$
cd snmpc
$ dir3

Directory SYS$SYSROOT:[PACKAGE,NMPC]

ASLREADME.FST;4       BIN.DIR;1       DIR160184,LIS;2
DIR170184,LIS;1       ICLIB,DIR;1     ISPOUT,SIM;8
L.OUT;2               LIBRARY.DIR;1    LOGIN.COM;71
LOGNAM.COM;12         MA160184,MA;1    MA170184,MA;1
MANUALS.DIR;1         SOFTGEN.DIR;1    SOURCE,DIR;1
SYSDEF.COM;1           TMP.DIR;1       UPDATE,DIR;1

Total of 18 files.
$
Directory USER\$DISK1:\ICL1

OREADME.TXT;1
LOG86.COM;30
LOGIN.COM;8
NMPC.DIR;1

86BUILD.DIR;1
LOGEASY.COM;66
LOGSYS.COM;16
OTOL.DIR;1

EDTINI.EDT;1
LOGEASY.COM;65
MAIL.NAI;1

Total of 11 files.

$
cd nmPC
$ dir3

Directory USER$DISK1:[ICL1.NMPC]

86SOFTDEV.DIR;1 LIB.DIR;1 READYSIM.DIR;1
SOFTGEN.DIR;1

Total of 4 files.
$ cd otol
$ dir3

Directory USER$DISK1:[ICL1.OTOL]

OTOL.C;8 OTOL.EXE;6 OTOL.OBJ;6

Total of 3 files.
$ cd 86build
$ dir3

Directory USER$DISK1:[ICL1.86BUILD]

OAREADME.TXT;1 86.LIB.DIR;1 AS86BLD.DIR;1
CC86BLD.DIR;1 LBR86BLD.DIR;1 LK86BLD.DIR;1

Total of 6 files.
$
APPENDIX C

Listings of "PRIME.C", "PRIME.S" Programs
#include <stdio.h>

int main() {
    int flags[size + 1];
    int i, odd, k, count, iter;

    for (i = 0; i <= size; i++) flags[i] = false;
    flags[2] = true;
    for (i = 0; (2 * i + 1) <= size; i++) flags[2 * i + 1] = true;

    /* printf("\nrange 1..%d \n", size); */
    /* printf("\n%d iterations\n", iterations); */
    for (iter = 1; iter <= iterations; iter++) {
        for (i = 3; i <= size; i++)
            if (flags[i])
                odd = i;
            for (k = 2; (k * odd) <= size; k++)
                flags[k * odd] = false;

        count = 0;
        for (i = 0; i <= size; i++)
            if (flags[i] == true)
                count++;

        /* printf("no %d prime found: \n", count); */
    }

    /* printf("\n%d primes, \n", count); */
}

cat primes

.text

    .extrn c$csv
    .extrn c$ret
    .extrn c$csv
    .extrn c$cssv
    .extrn c$tmp1
    .extrn c$tmp2
    .public _main

    _main: .segment code
    .assume cs:_main,ds:$common

brtosr: .set -42
    call 0,c$csv
    sub sr,$42
    mov -34[bp],t0

    .lineno 19
    L1:
    cmp -34[bp],+15
    js L3

    .lineno 19
    mov si,-34[bp]
    shl si
    lea bx,-32[bp]
    add si,bx
    mov [si],t0

    .lineno 19
    inc -34[bp]
    jmp L1

    .lineno 19
    L3:
    mov -28[bp],+1
    mov -34[bp],+0

    .lineno 21
    L11:
    mov bx,-34[bp]
    shl bx
    lea cx,1[bx]
    cmp cx,$15
    js L31

    *
L31:

.Lineno 21
mov si,-34[bp]
shr si
shr si
lea bx,-30[bp]
add si,bx
mov [si],#1

* .Lineno 21
inc -34[bp]
jmp L11

* .Lineno 21

L32:

.Lineno 26
mov -42[bp],#1
.
L12:

cmp -42[bp],#1
js L32

* .Lineno 26
mov -34[bp],#3

* .Lineno 29

L13:

cmp -34[bp],#15
js L52

* .Lineno 29
mov di,-34[bp]
shr di
lea cx,-32[bp]
add di,cx
cmp [di],#0
je L53
mov bx,-34[bp]
mov -36[bp],bx
mov -38[bp],#2

* .Lineno 34

L34:

mov ax,-38[bp]
mul -36[bp]
cmp ax,#15
js L53

* .Lineno 34
mov ax,-38[bp]
mul -36[bp]
mov si,ax
shr si
lea bx,-32[bp]
add si,bx
mov [si],#0

* .Lineno 34
inc -38[bp]
jmp L34

*
L32:
  .lineno 42
  mov  -40[bp],#0
  mov  -34[bp],#0
 *
L35:
  .lineno 44
  cmp  -34[bp],#15
  js   L55
 *
  .lineno 44
  mov  si,-34[bp]
  shl  si
  lea  bx,-32[bp]
  add  si,bx
  cmp  [si],#1
  jne  L75
  inc  -40[bp]
  jmp  L75
 *
L52:
  .lineno 26
  inc  -42[bp]
  jmp  L12
 *
L53:
  .lineno 29
  inc  -34[bp]
  jmp  L13
 *
L55:
  .lineno 52
  jmp1  0xc$cret
 *
L75:
  .lineno 44
  inc  -34[bp]
  jmp  L35
  _main:  .ends
  .end

$
APPENDIX D

N.mPc Installation Guide
This installation guide, listed as [21] in the references, has been added to explain the installation of the N.mPc package under the VMS operating system. The installation guide is a reprint from a paper delivered with the N.mPc system.
N.mPc under VMS *Preliminary*

Greg Ordy

Dept. of Computer Engineering and Science
Case Western Reserve University
Cleveland, Ohio 44106

ABSTRACT

This paper describes changes to the N.mPc System under VMS, the DEC operating system for the VAX family of processors. Described are the source release and the binary release.

1. Introduction

The N.mPc System was developed under the Unix operating system, and therefore can be expected to have some user interface changes under VMS. These changes fall into several categories.

*) Release Format
*) Use of VMS Logical Names/Mappings
*) VMS Foreign Commands
*) Program Option Mapping
*) Misc Notes

As a long term goal, we want to minimize all changes, and have a single N.mPc user interface across all host systems. At this first release of the VMS System, however, that is not completely possible.

As under Unix, we suggest a directory structure for the N.mPc system. All of the comments from this point on will use our directory structure. When following the hints, please modify the examples to reflect your system configuration. In particular, we have a login (UIC) named 'nmpc', which has a login directory named:

`dra0:[nmpc]`

Under this directory, we have several subdirectories. The structure is very similar to the Unix N.mPc system, and you should consult that documentation to get details on the structure. Of course our disks are named 'dra?', yours may well be different.
1.1. Release Format

The N.mPc System is distributed on 9 track tape in the VMS 'backup' format. Note, if you have a VMS Binary Distribution, the tape will be in ANSI format, as there is a single directory shipped. The single directory on the binary release should be placed in the directory named `[nmpc.bin]`.

In a Binary Distribution the 'library' is shipped as a Unix 'tar' image (basically) in the file 'library.tar'. The command 'rtar' is used to read this file and recreate the directory structure. To do this you must first create a directory under `[nmpc]` ([nmpc.lib]), then 'set def [nmpc.lib]', then 'rtar -vtxhf [nmpc.bin]library.tar'. You may then remove the file `[nmpc.bin]library.tar` as it consumes 2.6 Megabytes. There are ways of looking at the contents of the file without dumping the whole thing, reference the man page. The full library directory will be about 1.3 Megabytes after rtaring it. A Unix 'man' page is included to help you with the rtar command.

1.2. VMS Logical Names/Mappings

Logical names allow programs to refer to generic names which are mapped at execution time to a specific file. We use them to specify files which might have different names on different systems. Currently, the required logical names are:

```
define isplib dra0:[nmpc.bin]isplib.olb
define kernel dra0:[nmpc.bin]kernel.olb
define runhelp dra0:[nmpc.bin]runhelp.a
define ci dra0:[nmpc.bin]ci.exe
define motd dra0:[nmpc.bin]motd.txt
```

It is also necessary to have the DEC VAX-11 C compiler runtime library searched.

1.3. VMS Foreign Commands

It is necessary to run the N.mPc programs as 'foreign' DCL commands, so that the invocation options are properly passed to the programs. These mappings are most conveniently performed in your 'login.com' file. Foreign command mappings for N.mPc are:
1.4. Program Option Mapping

Under Unix, upper and lower case characters are treated separately by the command interpreter (shell). VMS, however, folds to a single case. Because some N.mPc commands use upper and lower case options, we were required to 'shuffle' some of the program invocation options. In the cases where a command had upper case options, without a matching lower case option, we simply made the option lower case. Note that VMS considers the default upper case, however the DEC Vax-11 C compiler switches cases, to mimic the standard Unix case. If there was a conflict, we had a choice of either moving the lower case option, and transferring the upper case option to lower case, or, we could leave the lower case option and remap the upper case to another lower case letter. In these cases we used a 'priority' system to move the option which is least used.

The following list should cover all of the mappings.

The micro assembler changes the upper case 'E' option to a 'b'. The upper case 'L' option is dropped.

Upper case options in the Interpreter (inter), mdump, and merge are switched to lower case. In addition, in as many places as possible, we have attempted to coordinate the options for specifying number base on output. The new standard is:

- decimal - 'd'
- hex - 'x'
- octal - 'o'

The linking/loader allocator (cater) has the most
changes, which are:

- A goes to a
- D goes to d
- d goes to z
- o goes to v
- O goes to o
- L goes to l
- F goes to f
- T goes to j
- m goes to l
- f goes to 2
- l goes to 3
- h goes to 4

The mas program maps upper case 'L' and 'I' to lower case.

The smp program maps the number base options to the new standard.

The ic program maps 'T' to 'd'.

When running a simulation under N.mPc there is a '-' option which causes simulated memory initialization information to be displayed. Under VMS, this option has been mapped to '-m'. Note however, that to use this, the simulation program itself will have to be defined as a foreign command, so that the option is correctly set up by DCL.

1.5. Misc Changes

The following paragraphs detail some additional changes to the system.

1.5.1. Message of the Day File

The N.mPc system has a message of the day file, whose contents are displayed whenever a simulation is started. There is a logical name mapping for the file, which logically is 'motd'. The delivered system has a motd file in the 'bin' directory, which has some hopefully cheerful message. This can be used to inform users of local changes to the system, and can be removed or zeroed out if desired.

1.5.2. New ISP' -k Flag

The ISP' compiler (ic) has a new invocation flag, '-k'. This flag (the keep flag) causes the source file with a '.lst' extension to remain after compilation. The compiler takes a source file with a '.isp' extension, and produces a file with a '.lst' extension, which contains a macro assembler source program which is the output of the compiler.
This file is assembled to produce a '.obj' object module, which is used by the Ecologist. Normally, the '.lst' file is removed by the compiler, however the '-k' flag will keep it around. The file can be used for compiler debugging.

1.5.3. Ecologist .sim or .obj Processors Allowed

The Unix ISP' compiler produces a '.sim' output file extension, whereas the VMS ISP' compiler produces a '.obj' file extension. To allow for 100% compatibility between the two systems, the VMS Ecologist will automatically map processor names in the topology file, and for a given root name, try to use either the '.sim' or '.obj' file, whichever is present. For example, if there exists a topology file with the declaration:

```
processor cpu = "xray.sim";
```

the VMS Ecologist will first try and open 'xray.sim', however if that fails, it will try 'xray.obj', which would be the actual name produced by the compiler, directly. The mapping goes either way.

1.5.4. Ecologist -t Flag

Although this flag was present on previous versions of the Ecologist (Unix based), it produces a slightly different output under VMS. Like the '-k' flag under VMS, the '-t' flag to the Ecologist causes it to NOT remove all of the temporary files it creates in the process of building a simulation. Under Unix, these temporary files are put in the '/tmp' directory, however under VMS, they are kept in the simulation directory.

For each ISP' output module, The Ecologist produces a '.dat' extension file, which is a copy of the '.obj' ISP' compiler output file, with some changes required by the Ecologist. The Ecologist also creates a file named 'EC*.Z', where the '*' represents a per simulation number, to uniquely identify the particular run of the Ecologist. This is an assembler source file, and is assembled to produce an 'EC*.OBJ' output module. The Ecologist also creates a VMS loader 'option' file, called 'LINKNMPC.OPT', which is used in conjunction with the '.dat' files, and the 'EC*.OBJ' file to build a simulation. The loader creates a 'LINKNMPC.STB' symbol table file, which is used by the Ecologist to create the standard N.mPc '.s' symbol file.

Normally, the Ecologist removes all of the above described temporary files. The '-t' option, however, will cause them to remain, again usually for debugging purposes.
1.5.5. Set Verify

Several pieces of the N.mPc produce VMS DCL command files, which if you have executed 'SET VERIFY' will produce output on your terminal. This is not a bug, and you should just ignore the data. (I try to).

1.5.6. Nmake Not Functional

The N.mPc 'nmake' program does not yet work under VMS. It should be up shortly. For those not familiar with its function, it is described in the Ecologist User's Manual.

1.5.7. No DCL Subshell in Runtime

The Unix N.mPc system allows a '"' runtime command, which causes the single line of text following the '"' to be interpreted as Unix Shell command. The corresponding DCL function is not yet implemented. To escape the simulation, one should use Control Y to suspend the simulation, and 'CONT' to continue.

1.5.8. Performance Improvements

We have discovered some areas of performance improvement, and can make an approximately 20% improvement in simulation speed, which has been done on the Unix/Vax N.mPc system. As soon as the basic VMS system is stable, these improvements will be migrated to VMS.

1.5.9. Ispline Runtime Command and FCVT

A very useful runtime command is 'ispline' which displays lines of ISP' source code as they are executed. To use this feature under VMS, it is necessary to have a copy of the ISP' source file which is in RMS stream format, rather than variable sized records, which is the type created by the editor. There is a new utility program called 'FCVT', which is in the N.mPc 'bin' directory, which performs this conversion. It accepts two arguments, a source file name, and a destination file name. It creates a copy of the source in the destination, creating a RMS stream file. Perhaps there is a way to do that under VMS, but it was easier for us to write a 10 line C program. In any case, you must ALWAYS specify a stream format file to the 'ispline' command, or it will not work. Locally, if we are going to do 'ispline' tracing, we copy the '*.isp' source file to a '*.ill' stream file, using FCVT. When we are running the simulation, we specify the '*.ill' file to the ispline command.
1.5.10. Ispline Runtime Performance Penalty

Although the ispline command is quite useful, it does really slow down a simulation, WHETHER OR NOT IT IS TURNED ON! So, if you don't want, or need statement tracing, you will get significantly faster simulations by turning off tracing in any or all ISP' source files. This is true for all implementations of N.mPc.

1.5.11. Nested Runtime Include Files

If you intend on using nested runtime include command files, you must use the 'fcvt' program to generate copies of the files with a stream RMS file type. For reasons similar to the ispline command, the simulation program must be able to perform random file positioning operations on the command file, and therefore the file type must be stream.

1.5.12. Ecologist Debug Option

The Ecologist accepts a new '-d' option, which links the VMS runtime debugger into the simulation.

1.5.13. Runtime Help Command

The runtime 'help' command is fully functional, and uses a file named 'runhelp.a' in the binary executable directory ([nmpc.bin]), although this can be easily changed as it is a logical mapping. In any case, this help file is in ascii readable format, which may be edited to reflect local changes. The format of the file is self explanatory, and will not be explained here. If you wish to change a particular help message, edit the file, move to the region which currently describes the command, and edit it.

1.5.14. Runtime Line Numbers

Several runtime command display the line an ISP' model is at when it is examined. The correct line numbers are not yet displayed.

1.5.15. VAX-11C Library

N.mPc is compiled using the DEC VAX-11C compiler. The Ecologist actually builds the simulation using the VMS assembler and loader, as well as the C runtime library. As a result, to build simulations, the VAX-11C runtime library must be installed in the proper directory, and your login.com file must cause the linker to search that archive. The library is named:

crtlib.olb

and is located in the [nmpc.bin] directory, as distributed.
This file must be placed in the VMS directory:

    sys$library

by the system manager (or whoever controls access to that directory). In addition, the command:

    def 1nk$library sys$library:crtlib.olb

MUST be in your login.com file to assure that the loader searches the C runtime library when building a simulation. If this is not done, the loader will generate a number of undefined reference messages when called from the Ecologist.

There is also an executable image named: 'yfork.exe', which is in the executable 'bin' ([nmtpc.bin]) directory. This program MUST be transferred to the SYSSSYSTEM directory on the VMS system. Note that this typically requires System Manager level of privileges. This program is used by the C runtime process creation functions, and if it is not placed in the proper directory (with execute access) simulations will hang when started, and will not run. These last two steps are only necessary if you do not have the DEC VAX-11 C Compiler.

1.5.16. New Post Processor

    There is a new, simple, post processor program for the Vax implementations of N.mPc. It is called 'pp'. A manual for it should be included in the delivery documentation package.

1.5.17. New Simulated Memory Listing Program

    There is a new program called 'mprint', which displays the contents of simulated memories. Documentation should be included in the delivery package.
The following command files illustrate the actual N.mPc installation on the VAX 11-780 at Communications Research Centre, Department of Communications, Government of Canada.
This command file is executed at log in from the SYSTEM login file SYLOGIN.COM

created 12/01/83 P Adamovits

corrected ( $ instead of "run") by Max Streit, Jan. 84

! DEFINE LNK$LIBRARY SYS$SYSDISK:PACKAGE.NMPC.BINJCTLIB.OLB
! DEFINE ISPLIB SYS$SYSDISK:PACKAGE.NMPC.BINJISPLIB.OLB
! DEFINE KERNEL SYS$SYSDISK:PACKAGE.NMPC.BINJKERNEL.OLB
! DEFINE RUNHELP SYS$SYSDISK:PACKAGE.NMPC.BINJRUNHELP.A
! DEFINE CI SYS$SYSDISK:PACKAGE.NMPC.BINJCI.EXE
! DEFINE MOTO SYS$SYSDISK:PACKAGE.NMPC.BINJMOTO.TXT
! DEFINE wttu tt:

HAS := $ SYS$SYSDISK:PACKAGE.NMPC.BINJMAS.EXE
HMERGE := $ SYS$SYSDISK:PACKAGE.NMPC.BINJMERGE.EXE
MDUMP := $ SYS$SYSDISK:PACKAGE.NMPC.BINJMDUMP.EXE
MICRO := $ SYS$SYSDISK:PACKAGE.NMPC.BINJMICRO.EXE
INTER := $ SYS$SYSDISK:PACKAGE.NMPC.BINJINTER.EXE
CATER := $ SYS$SYSDISK:PACKAGE.NMPC.BINJCATER.EXE
CMEM := $ SYS$SYSDISK:PACKAGE.NMPC.BINJCMEM.EXE
IC := $ SYS$SYSDISK:PACKAGE.NMPC.BINJIC.EXE
EC := $ SYS$SYSDISK:PACKAGE.NMPC.BINJEC.EXE
ECOLOGIST := $ SYS$SYSDISK:PACKAGE.NMPC.BINJECOLOGIST.EXE
SMP := $ SYS$SYSDISK:PACKAGE.NMPC.BINJSMP.EXE
SHE := $ SYS$SYSDISK:PACKAGE.NMPC.BINJSHE.EXE
NMAKE := $ SYS$SYSDISK:PACKAGE.NMPC.BINJNMAKE.EXE
FCVT := $ SYS$SYSDISK:PACKAGE.NMPC.BINJFCVT.EXE
PP := $ SYS$SYSDISK:PACKAGE.NMPC.BINJPP.EXE
MPRINT := $ SYS$SYSDISK:PACKAGE.NMPC.BINJMPRINT.EXE
DEFINITIONS

```$ def/proc c8086 sys$susdisk:\{package.c8086\}
$ def/proc otol user$disk:\{ic11.otol\}
$ def/proc 86build user$disk:\{ic11.86build\}
$ def/proc 86lib user$disk:\{ic11.86build.86lib\}
$ def/proc as86bld user$disk:\{ic11.86build.as86bld\}
$ def/proc cc86bld user$disk:\{ic11.86build.cc86bld\}
$ def/proc lbr86bld user$disk:\{ic11.86build.lbr86bld\}
$ def/proc 1k86bld user$disk:\{ic11.86build.1k86bld\}
```

```
C C 8 0 8 6 C O M M A N D S

$ cr1 \(==\) $ user$disk:\{ic11.86build.cc86bld\}cr1.exe
$ cr286 \(==\) $ user$disk:\{ic11.86build.cc86bld\}cr286.exe
$ cr386 \(==\) $ user$disk:\{ic11.86build.cc86bld\}cr386.exe
$ cpp \(==\) $ user$disk:\{ic11.86build.cc86bld\}cpp.exe
$ asm86 \(==\) $ user$disk:\{ic11.86build.as86bld\}asm86.exe
$ 1br86 \(==\) $ user$disk:\{ic11.86build.1br86bld\}1br86.exe
$ 1dr86 \(==\) $ user$disk:\{ic11.86build.1k86bld\}1dr86.exe
$ 1kr86 \(==\) $ user$disk:\{ic11.86build.1k86bld\}1kr86.exe
$ otol \(==\) $ user$disk:\{ic11.otol\}otol.exe

D E V I C E A S S I G N M E N T

Assignment is necessary in order to tell the assembler
where the hash table is located. (ASM86 believes that
the table is in DUAO:\{LANTECH.AST0SEND\}AS8086.SYM)

$ assign drbl: dua0:
```
SPECIAL LOGIN.COM('LOGEASY.COM)

The purposes of this login file are as follows:

1. Set up directory names and paths for the N.mPc simulation packages.
2. Set up directory names for miscellaneous uses.
3. Define all the UNIX commands in VMS.

Max Streit, January 84
Claude Laferriere, February 84

*************************************************************************

DIRECTORIES ON SYS$SYSDISK:
$ def/proc snmPc sys$sysdisk:[rackade.nmPc]
$ def/proc version1 sys$sysdisk:[rackade.c8086.version1]
$ def/proc version2 sys$sysdisk:[rackade.c8086.version2]

DIRECTORIES ON USER$DISK1:
$ def/proc ic12 user$disk1:[ic12]
$ def/proc ic13 user$disk1:[ic13]
$ def/proc ic14 user$disk1:[ic14]
$ def/proc trxsim user$disk1:[ic14.trxsim]
$ def/proc persim user$disk1:[ic14.persim]
$ def/proc compsim user$disk1:[ic14.compsim]
$ def/proc cpcom user$disk1:[ic14.cpcom]
$ def/proc isplibr user$disk1:[ic14.isplibr]
$ def/proc Oreadme user$disk1:[ic14.0readme]
$ def/proc ic11 user$disk1:[ic11]
$ def/proc nmPc user$disk1:[ic11.nmPc]
$ def/proc lib user$disk1:[ic11.nmPc.lib]
$ def/proc 86softdev user$disk1:[ic11.nmPc.86softdev]
$ def/proc readysim user$disk1:[ic11.nmPc.readysim]
$ def/proc softgen user$disk1:[ic11.nmPc.softgen]
$ def/proc as86 user$disk1:[ic11.nmPc.86softdev.as86]
$ def/proc c86 user$disk1:[ic11.nmPc.86softdev.c86]
$ def/proc 11ib user$disk1:[ic11.nmPc.86softdev.11ib]
$ def/proc sbr9989 user$disk1:[ic11.nmPc.readysim.sbr9989]
$ def/proc 86val user$disk1:[ic11.nmPc.readysim.86val]
$ def/proc 86sieve user$disk1:[ic11.nmPc.readysim.86sieve]
UNIX COMMANDS

$ cd := set def
$ pwd := show def
$ rm := delete
$ cp := copy
$ cat := type
$ mv := rename
$ ls := dir
$ dir3 := dir/col=3
$ who := show users
$ mkdir := create/directory
$ time := show time
$ od := dump
$ ps := monitor process
$ pstat := monitor process/state
$ man := help

SPECIFIC VMS COMMANDS

$ set control=t
$ stat := monitor states
$ sus := show system
$ mem := show memory
$ proc := monitor processes