University of Wollongong

Research Online

Department of Computing Science Working Paper Series

Faculty of Engineering and Information Sciences

1983

A screen oriented simulator for a DEC PDP-8 computer

Neil Gray
University of Wollongong, nabg@uow.edu.au

Follow this and additional works at: https://ro.uow.edu.au/compsciwp

Recommended Citation

Gray, Neil, A screen oriented simulator for a DEC PDP-8 computer, Department of Computing Science, University of Wollongong, Working Paper 83-2, 1983, 65p.
https://ro.uow.edu.au/compsciwp/69

Research Online is the open access institutional repository for the University of Wollongong. For further information contact the UOW Library: research-pubs@uow.edu.au

THE UNIVERSITY OF WOLLONGONG

DEPARTMENT OF COMPUTING SCIENCE

A SCREEN ORIENTED SIMULATOR FOR A DEC PDP-8 COMPUTER

N.A.B. Gray

Department of Computing Science

University of Wollongong

Preprint No 83-2 **January 25, 1983**

P.O. Box 1144, WOLLONGONG, N.S.W. AUSTRALIA
telephone (042)-282-981
telex AA29022

A Screen Oriented Simulator for a DEC PDP-8 Computer.

N.A.B. Gray.

Department of Computing Science, University of Wollongong, PO Box 1144, Wollongong NSW 2500, Australia.

ABSTRACT

This note describes a simulator for the DEC PDP-8 computer. The simulator is intended as an aid for students starting to learn assembly language programming. It utilises the simple graphics capabilities of the terminals in the department's laboratories to present, on the terminal screen, a view of the operations of the simulated computer.

The complete system comprises two versions of the program for simulating a PDP-8 computer and a simplified "assembler" for preparing students' programs for execution. There are also a number of example PDP-8 programs illustrating particular aspects of that computer.

The first version of the simulator is intended to help illustrate a conventional computer's **fetch-decode-execute** cycle. In this version, there is a three part display. The three parts represent (i) the central processing unit ("cpu"), (ii) the communications path joining the cpu and main memory ("bus") and (iii) a window into main memory. These displays allow a detailed presentation of how a program is actually executed on a computer. Data, both instructions and program data, can be seen being read out from memory and being transferred over the bus to registers in the cpu. There instructions are decoded and program data manipulated. Results from computations can be seen being transferred back out of the cpu registers, over the bus, to memory where they overwrite previous values. The simulation may be run continuously, at a user selectable speed, or may be set to pause, and await user response, between each of the stages of the machine's instruction cycle.

A second, more elaborate version of the simulator program, provides an environment for introducing basic concepts of assembly language "debugging". This version also provides displays of the simulated machine's cpu and memory. The level of detail of these displays is user selectable. It incorporates a fairly conventional "debugging" function that allows users to run their programs in a controlled manner. Users may, for example, specify "breakpoints" in the PDP-8 programs that they have prepared for simulated execution. On reaching such a breakpoint, execution of a program is suspended temporarily to allow inspection of the contents of the cpu registers and of the memory of the simulated machine.

This more advanced version of the simulator program provides a reasonably realistic model of how input and output ("I/O") are performed on small mini- and micro-computers. The simulated machine is equipped with some standard peripherals such as a clock and analog-digital converter. These simulated peripherals can be operated using either "flag-driven" I/O or through an "interrupt" mechanism.

1. Introduction

In their second year of Computing Science, students at the University of Wollongong are required to take a course introducing machine organization and assembly language programming. Prior to taking this course, students' sole computing experience is a one year introductory course on programming, using the PASCAL language on a time-shared system. For students continuing in computing science, the "assembly language" course provides a grounding for subsequent studies on compilers, operating systems and more advanced courses on micro-computers ("micros"). The major benefit for general students is a wider perspective on computers. A basic understanding of how computers work, and how they may communicate with peripheral devices, is particularly advantageous to students in the physical sciences who may later need to interface computers with their experiments.

Such an introductory course on machine organization and assembly language involves first some general overview of the organization of conventional computers and then some revision of various number representations and logical operations (so that students will not experience any difficulties due to unfamiliarity with binary, octal or hexadecimal representations of data in a computer). After completing this introductory part of the course, students will proceed to programming in the "assembly language" of some particular computer. This involves learning new programming constructs. The high-level PASCAL constructs for procedure calls and conditional statement execution are considerably removed from the detailed operations that can be performed directly by a computer. Consequently, students must change their programming styles from PASCAL to the regime of the new machine's assembly language. Furthermore, students must learn totally different methods for localizing and identifying errors in their programs.

Such an introductory course on assembly language programming could be developed around the use of small micro-computers by the students. It is quite possible to use a time-shared computer to prepare programs for micros, and to transfer these programs to connected micros for trial execution. This is indeed the scheme employed in the more advanced third year "*Microcomputers*" course as taught in the department. Although such "hands-on" experience might well be advantageous to students, departmental resources preclude this approach being used with the large second year classes. Instead, the introductory course in machine organization and assembly programming must rely on the resources of the department's time-shared Perkin-Elmer computers running under the UNIX operating system.

In previous years, the course has emphasized the use of the Perkin-Elmer machines and has, in large part, been an exposition on how to write large assembly language programs for a complex machine with a sophisticated operating system. There are a number of disadvantages relating to the use of the Perkin-Elmer machines when first introducing assembly language programming. These machines are complex, (baroque?), in their architecture. Students are immediately confronted with a plethora of instructions, data formats and programming conventions. The UNIX system does provide some aids for debugging assembly language programs, specifically the **adb** interactive debugging program, but these aids are themselves complex

and difficult to learn. Of course, because the students' programs must be run under the general time-sharing regime, all input and output must be performed through "magical" calls to the operating system. If all I/O tasks are thus delegated to the operating system then it becomes more difficult for students to gain any appreciation of how input and output are actually performed; such an appreciation is an essential prerequisite to subsequent studies of operating systems.

In 1982, an attempt was made to find a simpler environment in which the basic concepts of assembly language programming might be introduced. Mr. R. Nealon, the software Professional Officer in the department, had previously developed a simple screen-oriented simulator for a hypothetical machine, the "r80". This simulator runs on the department's time-sharing system and utilizes the limited graphics capabilities of the standard terminals. The simulator system allows for programs to be written in the assembly language of the r80 and then "visibly executed". The r80 has a relatively small memory and only a couple of registers in its cpu. All elements of the machine, both cpu registers and memory, can be simultaneously displayed on the terminal screen. This display can show how the execution of each instruction changes the state of the simulated machine. The r80 simulator allows programs to be executed normally or, in "single-step" mode, one instruction at a time. This r80 simulator was adopted and used in both the lecture course and in the first two assembly language assignments. Most students found the display of the r80 executing their programs to be of considerable assistance in obtaining some understanding of how a computer operates. Only after these initial assignments had been completed did students move on to the greater complexities of assembly language programming for the Perkin-Elmer machines.

The r80 is a "hybrid" combining features present in many current micros; its mechanisms for addressing memory, for calling subroutines and for manipulating different sized data elements are, however, somewhat unconventional. The r80 design does not attempt to represent any realistic I/O mechanism. Apart from the single-stepping facility, the simulator does not provide any debugging aids akin to those that students must employ in more advanced exercises. Although of considerable value in initial exercises, the overall applications for the r80 were limited by these features and by the small size of memory available for programs. (The r80 has only four hundred bytes of memory and each instruction requires four bytes).

It was decided to try to devise a more elaborate computer simulator starting from the concepts Nealon had developed and expressed in the r80. The objectives of the new simulator included (i) provision of more memory to allow for larger programs, (ii) incorporation of some conventional debugging mechanisms that could be used in association with visual displays of the simulated machine, (iii) implementation of some scheme for illustrating program execution at varying levels of detail down to the individual micro-program steps of the computers instruction cycle, and (iv) realistic simulation of I/O. Hopefully, this more elaborate simulator will allow students to learn more aspects of assembly language programming within a controlled and helpful environment and may possibly allow students to write simple I/O programs without in anyway adversely impacting the real computer's time-share system.

Given the decision to implement a simulator, one can then of course choose any real or hypothetical computer to simulate. The new simulator that has been developed is based closely upon the Digital Equipment Corporation's (DEC's) PDP-8 computer, possibly the first widely used mini-computer. This almost archaic machine is of course extremely limited in capacity and unrepresentative of modern mini- and micro-computers. However, the limitations of the PDP-8 are of little consequence for this initial teaching application.

The PDP-8's instruction set is very sparse; frequently many PDP-8 instructions are necessary to realize the same effect as can be obtained by a single instruction on

a more sophisticated machine. The available instructions are however quite sufficient for the assignments attempted by students. Larger instruction repertoires tend to confuse by offering many alternative mechanisms for attaining the same objective.

On the PDP-8 all data elements, program data and instructions, are constant in size. Much greater flexibility can be attained on more modern machines with bit, byte, half-word, full-word and double-word data elements; however, this very flexibility entails artificial problems of data-alignment that are frequently difficult to comprehend when first beginning assembly programming. The subroutine call mechanism, and interrupt handling mechanism, on the PDP-8 are both relatively simple. This very simplicity is inconvenient in sophisticated applications but, until students are familiar with the simple approaches (and their limitations), more elaborate mechanisms frequently seem both arbitrary and over complex.

The Digital Equipment Corporation manufactured several variants on the basic PDP-8 machine. These differed not only in their actual hardware realization but also, to minor degrees, in their instruction repertoire. Later models tended to have rather more instructions and some had an extra register in their cpus. The simulator does not attempt to capture any specific model of the PDP-8. It is closest to one of the earlier variants.

Both the simulator programs and the assembler are written in standard PASCAL. The sources for these programs are available to students, and components of the code are used in illustrative examples during the course.

The rest of this document consists of notes for students. Topics covered include the following:

a) An overview of the PDP-8. (This is a somewhat cursory review of conventional computer organizations, using the PDP-8 as a specific example. This material should have been covered, in both greater breadth and detail, in lectures prior to students starting to use the simulators).

b) An introduction to the basic simulator. (This section explains the form of the display in that version of the simulator used to illustrate the "fetch-decode-execute" cycle of the machine. The concept of an "object" file containing a machine compatible representation of a program is introduced, as are assumptions about where programs are placed in the memory of the machine).

c) Preparation of programs for the simulator. (A simple assembler program is introduced. This is a standard two-pass assembler producing absolute code. The basic instruction set of the PDP-8 is presented and a simple example program, such as students might be expected to write, is given).

d) Addressing mechanisms of the PDP-8 and subroutine calls. The "paged" addressing mechanism of the PDP-8 is covered in a little more detail, the early program examples avoid the addressing problems by using only page 0 and page 1. The subroutine call mechanism is illustrated.

e) Debugging assembly language programs --- the advanced simulator. (The advanced version of the simulator is introduced together with its "debugging" functions. These functions implement another variant on DDT, adb or other similar debugging systems. The various display options of the advanced simulator are covered).

f) Flag-driven I/O. Both the sections on I/O are considerably abbreviated versions of material covered in the lecture course. Input and output are presented here first in terms of "flag-driven" I/O methods. The advanced simulator has a "pseudo-keyboard" and a "pseudo-teletype" which are in reality standard files on the UNIX system. Data can be read from and written to these pseudo-devices by standard I/O instructions of the PDP-8. The simulation is realistic save that, for obvious reasons, the speed of these slow peripheral devices has been increased by

somewhat more than three orders of magnitude.

g) Interrupts. Interrupt driven I/O is introduced using an example program that "acquires and processes" data from a pseudo-analog/digital converter. The data acquisition rate is constant, being clock driven; the processing time necessary for each data element varies (data are random numbers, the processing really consists of counting the number of binary 1s etc). In the long run the acquisition and processing rates are approximately balanced but there are short term fluctuations making it necessary to buffer data between acquisition and processing.

h) Limitations of the PDP-8 architecture. A few of the limitations of this computer are briefly noted.

2. The DEC PDP-8 Computer.

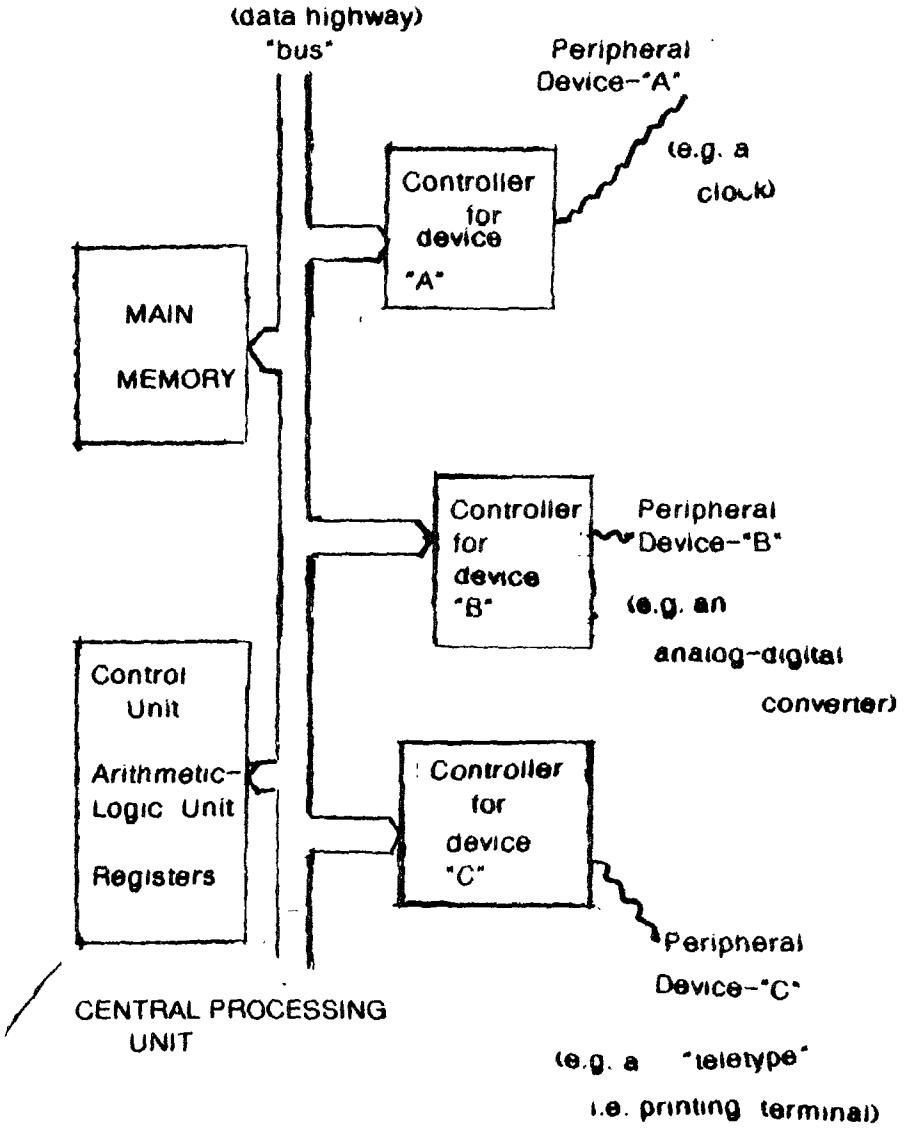
The PDP-8, circa 1965, is an early model laboratory computer. Typically, it is used for tasks such as monitoring simple laboratory experiments or running remote-job-entry stations for larger computers. The processor is still manufactured and used as the basis of certain rather restricted word-processing systems; it is also used in one of DEC's less sophisticated "personal" computers.

A considerably simplified representation of the machine is shown below. We can describe the machine in terms of four main components.

First, there is the computer's main memory where both program and data are stored.

Second there is the central processing unit. The cpu contains three subsystems --- a control unit which effects the execution of the program, an arithmetic logic unit which modifies data and a set of registers. The cpu registers hold data currently being used such as, for example, the last character read in from a terminal or the current temporary result of some computation.

The third main component, the "bus", joins the cpu to memory (and to peripheral devices). The bus can be viewed as a communications highway consisting of many signal wires. Some of these signal wires carry control signals, others convey individual bits of data and still others specify the destination of the data being transferred on the bus.

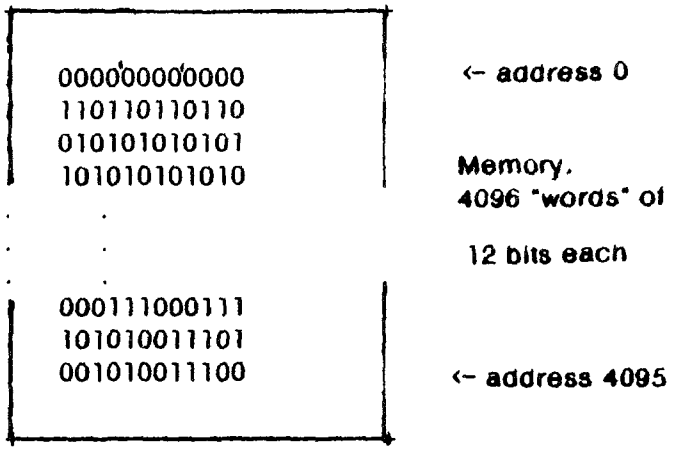


The fourth and final component of our system is comprised of the various peripheral devices and their controllers. Data can be sent to or received from peripheral devices. For each such device there will be a controller. A controller mediates between the device itself and the computer's bus; each controller will have specially designed circuitry to convert data from their external form (e.g. a slow sequence of electrical pulses from something like a keyboard or a particular transient voltage on an analog to digital converter ("a/d")) into the conventional signals used within the computer.

It is necessary to know something of the internal structure of main memory and of the cpu; details of the bus and device controllers are less important.

2.1. The Main Memory.

The main memory of a PDP-8 computer comprises 4096 "words". (Our simulated machine has less, real PDP-8s could be extended to have more memory through various hardware "kludges"). It is convenient to regard memory as being a vector, i.e. one-dimensional array, so a PASCAL data structure declaration for the memory would read something like "**const coresize = 4095; var store : array[0..coresize] of word**". The index number of each word in this memory is referred to as its "address" or "location".

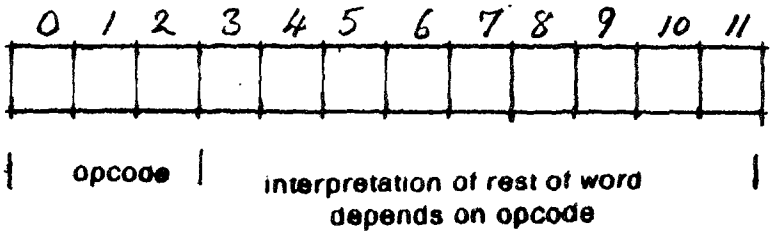


MAIN MEMORY

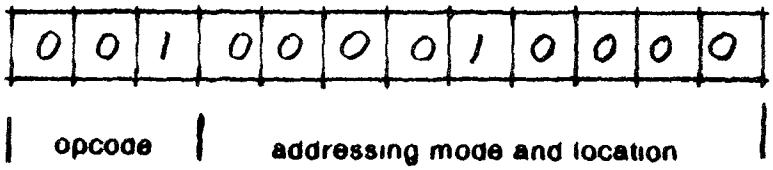
On the PDP-8, memories were usually constructed out of magnetic cores; modern variants of the machine use semi-conductor memories. It takes a finite time to write data into a word of memory or to read data out of a word. Typically, the "memory cycle time" is about one micro-second (i.e. one one-millionth of a second) with semi-conductor memories being somewhat faster. This memory cycle time was the major factor in determining the speed of execution of programs. As will be explained in more detail later, the execution of any individual instruction on a PDP-8 involved from one to three memory accesses. Consequently, with a one micro-second store, the machine could execute something around 300,000-500,000 instructions per second.

Each word in a PDP-8's memory held 12 bits. A word could be regarded as holding an instruction, an unsigned number in the range 0-4095, an address (which was of course just an unsigned number in the range 0-4095), a signed number in the range -2048..+2047, one eight bit character (with four bits unused) or two six bit characters packed together. The data in any word are of course just some particular patterns of twelve 1s or 0s. The interpretation of these binary patterns depends solely on how they are used by the program. The following are examples of different binary patterns and their alternative interpretations:

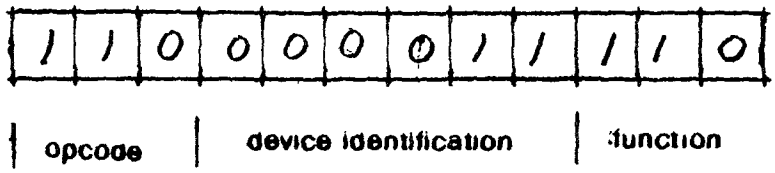
2.1.1. Instructions)



e.g.

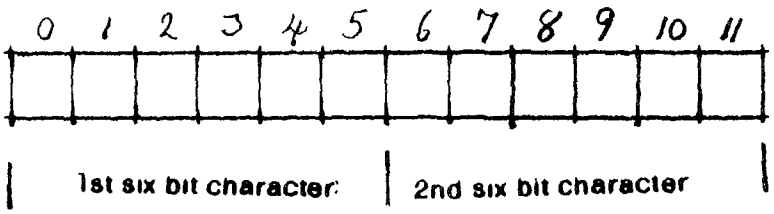


binary value 001000010000
octal value 1 0 2 0
interpretation as an instruction: TAD 20
meaning add the contents of location 20 to the accumulator register.

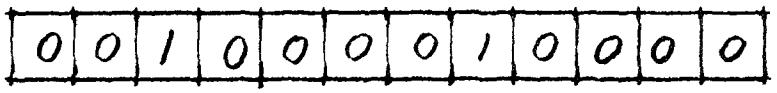


binary value 110000011110
octal value 6 0 3 6
interpretation as an instruction: KRB
meaning read the contents of the keyboard buffer into the accumulator and clear keyboard flag.

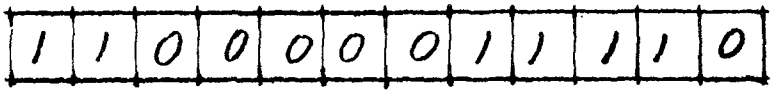
2.1.2. Characters)



e.g.



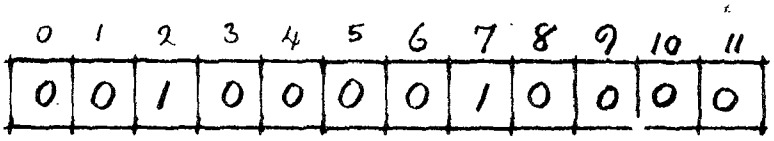
binary value 001000010000
octal value 1 0 2 0
interpretation as characters: HP



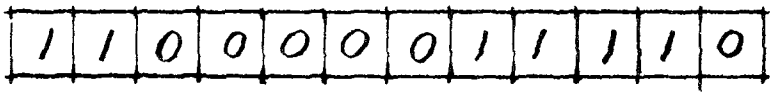
binary value 110000011110
octal value 6 0 3 6
interpretation as characters: 0^

2.1.3. Unsigned Numbers)

e.g.



binary value 001000010000
octal value 1 0 2 0
interpretation as a number: octal 1020
1x512 + 0x64 + 2x8 + 0x1
512 + 16
=528 decimal

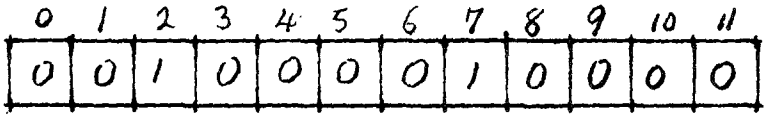


binary value 110000011110
octal value 6 0 3 6
interpretation as a number: octal 6036
6x512 + 0 x64 + 3x8 + 6x1
3072 + 24 + 6
=3102 decimal

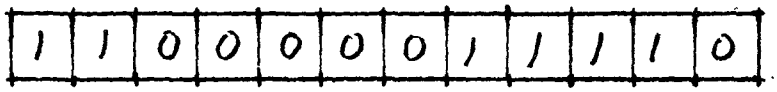
2.1.4. *Two's complement signed numbers)*

(As will be discussed in lectures, there are several different conventions regarding how negative numbers should be represented in a computer. The PDP-8 uses *two's complement notation* as do most, but by no means all other modern computers).

e.g.



binary value 001000010000
octal value 1 0 2 0
interpretation as a number 528 decimal

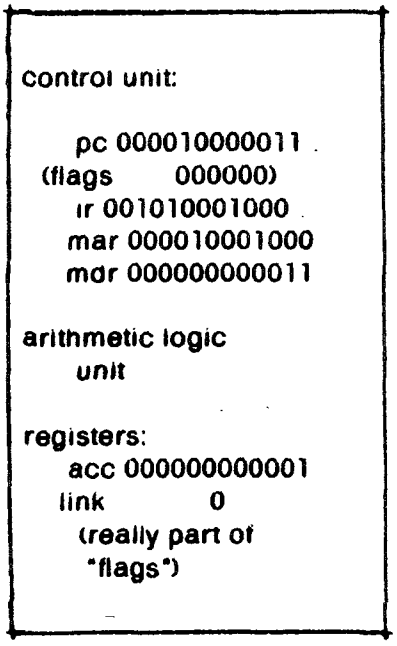


binary value 110000011110
octal value 6 0 3 6
interpretation as a number 6036octal
is 2's complement of 1742octal
i.e -(1x512 + 7x64 + 4x8 + 2x1)
-(512 + 448 + 32 + 2)
-(994)

2.2. The Central Processing Unit (CPU).

As noted earlier, there are three subsystems within the central processing unit. These three subsystems being the **control unit**, the **arithmetic logic unit** and the **registers**. The simplest of these subsystems is the "registers" subunit.

CENTRAL PROCESSING UNIT

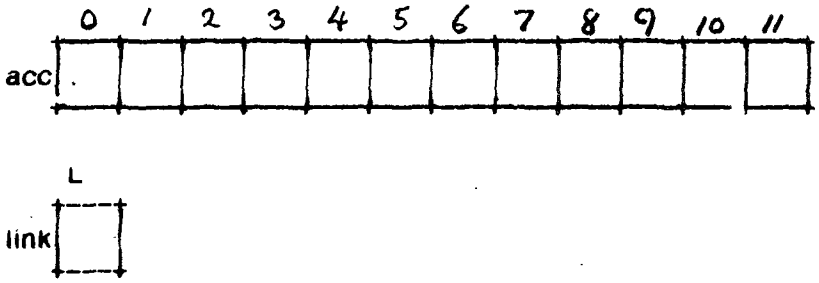


2.2.1. The registers.

Data being manipulated by a program are held in the registers. Some, or all, of the inputs to and outputs from the arithmetic logic unit must be routed via registers. A computer's "registers" constitute a kind of smaller but much faster version of main memory. The time needed to access data in a register will typically be less than a tenth of the time needed to access data in main memory. Most machines have several registers, anything from four to sixteen. Registers are used to hold those data that are of greatest import at any one stage in a program. Typically, they are used to hold counters for "for-loops", temporary results from computations, indices (subscripts) for arrays.

Some machines build restrictions into register usage so that, for example, particular registers can only be used as array indices. The task of the assembly language programmer, or of the compiler processing a high level language program, is to produce code such that efficient use is made of the available registers. Life is simpler on the PDP-8, there is only one data register --- the accumulator (acc). (The "link" can be regarded variously as an extra 1-bit register, or as an extension of the accumulator or as one bit of the "flags" register in the control unit (see below)).

The PDP-8's Registers:



The PDP-8's one twelve-bit "acc" register has to be used for all aspects of a computation. For example, in a simple program loop summing the elements of an array the "acc" is employed as follows. First, the loop index is loaded into the acc and "compared" with the loop limit. (Since there is no "compare" instruction, this operation in fact entails a sequence of instructions to compute the difference between current and limit values for the loop index. This difference is left in the acc: the difference will be negative if the loop has not terminated). If the loop has not terminated (as determined by testing for a positive or negative value in the acc), then the loop index is again loaded into the acc and combined with the address of the start of the array to derive the address of the particular element required. This computed address is then stored in some temporary location in memory. Then, making use of the address just calculated and stored, the required data element is loaded into the accumulator. The running total is added in and the result stored back into memory. Finally, the program would have a loop back to the point where the acc was again loaded with the loop index for the termination test. Obviously, when only a single register is available to perform each and every one of these tasks, much of the program will comprise code to load and store values from that register.

2.2.2. The Arithmetic Logic Unit (ALU).

Data are only manipulated in the arithmetic logic unit. This unit contain many specialized circuits. There will for example be an "adder" circuit that can add together two binary numbers; other circuits will perform boolean operations such as "anding" or "oring" together two binary patterns. More costly computers incorporate more circuits in the arithmetic logic unit; there could be a circuit for multiplying integers or even circuits capable of processing "real" numbers.

One of the features by which computers can be differentiated is the flexibility of the mechanisms for feeding data into the arithmetic logic unit and for directing the results into storage. The Data General NOVA computer is an example of a machine with a simple but restrictive mechanism. On the NOVA, only data in high speed registers can be passed to the arithmetic logic unit and the results must be returned in one of the input registers. More typically, machines allow also data fetched from memory to be combined with data in a register with the result being placed in that register. Some machines are still more flexible.

The PDP-8 has the simplest and most restrictive form of arithmetic logic unit. There are really only two circuits --- an adder, and a circuit for performing a boolean

"and" operation. These circuits combine data in the acc with data fetched from memory leaving the results in the acc.

2.2.3. The Control Unit.

The final component of the cpu is the control unit. This comprises a number of registers containing information that define the state of execution of a program along with circuitry that identifies what instruction must be performed next and how to perform that instruction.

2.2.3.1. The Control Unit's Registers.

```
    pc 000010000011
(flags      000000)
    ir 001010001000
   mar 000010001000
   mdr 000000000011
```

Memory Address Register ("mar") and Memory Data Register ("mdr").

The control unit registers with the simplest applications are the memory address register, "mar", and the memory data (buffer) register, "mdr" ("mbr"). The role for these registers is to interface between the cpu circuits and the bus. If the cpu wants some datum, either a program datum or an instruction, then it loads the address of the memory location containing the required datum into the mar and puts a "read memory" signal onto the control lines of the bus. The memory unit responds by going to the location defined by the value in the mar register, fetching a copy of the contents of that location and returning this copy, over the bus, to the mdr. Once the datum has thus been obtained, the cpu circuits route it internally either to the instruction register, "ir", if it is to be interpreted as an instruction, or via the arithmetic logic unit through to the accumulator if the datum represents something that the program is to manipulate. Data being written to main memory also pass via the mdr and thence onto the bus. Similar conventions pertain when transferring data between the cpu and peripheral device controllers.

Instruction Register ("ir").

The instruction register holds the bit pattern representing the instruction currently being executed. Through their programming experience in PASCAL, students are familiar with sort of macroscopic "instructions" such as the additions/multiplications etc involved in expression evaluation, assignment "instructions" and procedure call "instructions". When programming at assembly language level, students have to learn to define the corresponding operations at a more microscopic level where, for example, even a simple PASCAL assignment statement/"instruction", e.g. `a:=b;`, expands into three machine-level instructions (viz clear the acc, add contents of memory location "b" to the acc, store the contents of acc in memory location "a"). Execution of a program involves i) **fetching** each successive instruction into the ir register, ii) **decoding** the fetched instruction to determine exactly what it specifies, iii) **execution** of the specified sequence of data manipulations.

Program Counter ("pc").

Another concept, already developed through the programming of PASCAL loops and conditional statements, is that of the "locus" of control. Students are familiar with the idea of something --- "the computer" --- stepping through a sequence of PASCAL statements comprising a program. Physically, this locus is realized in the form of the program counter "pc". The contents of the "pc" register is the address in memory containing the next instruction to be executed. For straight-line code the program counter can simply be increased after each instruction is fetched so that it contains the next instruction address. (On the PDP-8, all instructions occupy exactly one word of memory each, so that for straight-line code the pc can be incremented by 1 each time).

Transfers of control, when encoding loops or for jumping around code that is only conditionally executed (as in *"if ... then begin ...; ...; ... end"* etc), are a bit more complex. Basically, the address of the next instruction to be executed has to be calculated, or retrieved. This computed address is loaded into the pc.

Subroutine calls, (procedure calls), are even more tiresome. When calling a subroutine it is not sufficient merely to transfer control to that bit of code (as achieved by loading the pc with the address of the subroutine); somehow a mechanism must be provided to get back to the main calling procedure, resuming execution at that instruction immediately following the subroutine call. The mechanisms that are provided for adjusting the pc across subroutine calls constitute another of the more obvious ways for differentiating between various designs of computers. The PDP-8 adopts a peculiarly crude approach adequate only for the simplest applications. This subroutine call mechanism is detailed later.

Flags resister.

Another typical constituent register of the control unit in a cpu is the "flags" register. This comprises a set of one-bit flags each indicating various status settings. Some of these flags might record the results of previously performed comparison operations on machines with explicit compare instructions. Others might detail information regarding the status of the bus and its use by peripheral devices. The PDP-8, at least its early variants, does not really possess such a "flags" register.

2.2.3.2. Instructions, their format and decoding.

On the PDP-8, the formats of instructions are simple. Three bits of an instruction word, bits 0-1-2, identify the actual instruction to be performed. With three bits it is possible to represent eight different binary patterns, viz 000, 001, ..., 111 (or 0-7 octal). Correspondingly, the machine has eight basic instructions. The different binary patterns, interpreted as representing instructions, are referred to as "**opcodes**".

One of these eight basic instructions, "iot", is used to specify control signals for peripheral devices. Consideration of iot instructions is deferred until later.

Six of the remaining seven instructions are basically similar in form. These are the "memory reference" instructions. They use the nine remaining bits of the twelve-bit instruction word to identify a memory location. This location might constitute the source from which data are to be fetched when performing an addition or an "and" operation, or might represent the destination into which the current contents of the accumulator register are to be copied. In a "jmp" (i.e. goto) instruction, the address bits of the instruction word will specify the memory location containing the next instruction.

The final "instruction", "opr", really comprises two whole families of instructions for manipulating data in the acc and link registers. Some of these "operate" instructions involve clearing (i.e. setting to zero) the acc and/or link registers, complementing

the bits in these registers (all binary 1s become 0s and vice versa) and rotating the bit patterns around.

Other operate instructions implement a rather restricted form of conditional branch instruction. These "skip" instructions are limited in that they allow the program to branch around only one instruction! A typical skip instruction, "sna" (skip on non-zero accumulator), involves testing to see if the acc is non-zero, if so the program counter will be incremented causing the immediately succeeding instruction to be skipped over. In the same group as the skip instructions there is the "hlt" (halt) instruction; this stops the computer at the end of a program.

2.2.3.3. The Fetch-Decode-Execute Cycle.

The circuits both for identifying the instruction to be performed and for performing instructions can best be conceived in terms of a stored "program". (In fact, that is quite often how the circuits are implemented). This "program" defines a sequence of data transfers between specified registers. This control program can be envisaged as being something of the the form:

```
repeat
  fetchinstruction;
  decodeinstruction;
  executeinstruction;
until halted;
```

The "halted" flag gets set when the machine executes a "halt" instruction.

The individual procedures, *fetchinstruction*, *decodeinstruction*, *and executeinstruction*, will comprise code that specifies how data is to be transferred between various registers of the cpu and locations in memory. Thus, *fetchinstruction* could be something like:

```
procedure fetchinstruction;
begin
    { Copy program counter to mar                }
    mar:=pc;
    { Send request to memory, via bus, for contents }
    { of location specified by mar               }
    frommemory;
    { Copy retrieved datum from mdr to ir        }
    ir:=mdr;
    { Increment pc so that its pointing at next  }
    { instruction.                               }
    { i.e. pass 1 & contents of pc to adding circuit }
    {    truncate the result to 12-bits          }
    {    store truncated result back in pc       }
    pc:=add(pc,1) mod 4096;
end;
```

The procedure for decoding an instruction consists, primarily, of a big "*case*" statement. The three bits, 0-1-2, containing the opcode must be abstracted from the ir register. This opcode, 0-7 octal, defines the branch of the case statement appropriate for the particular instruction to be executed. Thus, opcode 0 signifies that an "and"

instruction is needed, while opcode 7 specifies an "opr" operate instruction.

The interpretation of the remaining bits of the instruction word depends on the particular instruction being executed. For instructions like "and", "tad" (i.e. add), "jmp" (i.e. jump or goto) and "dca" (i.e. deposit contents of acc in memory and then clear acc) the remaining bits in the ir register will specify, directly or indirectly, the address of the memory location to be used. For these instructions, the rest of the decoding process consists of resolving exactly what memory address is being referenced and getting this address into the mar register. In an "iot" instruction, the remaining bits will specify which device controller is to receive a command signal and, also, will identify the particular command signal that must be sent. The remaining nine bits in an operate instruction specify the particular bit-manipulations or skip-tests that must be performed on the contents of the acc and link.

Finally, once the instruction has been fully decoded it must be executed. Execution can again be described in terms of a program specifying transfers between registers. A simple example is provided by the PDP-8's "dca" instruction. The effect of this instruction is to store the current contents of the acc in a specified memory location and to clear the acc. The instruction decoding process will have identified the instruction as being "dca" (from its particular opcode 011-binary 3-octal), and will have decoded the address bits to derive the required address which will be held in the mar register. The actual execution of the "dca" instruction can be defined in terms of the following micro-program for manipulating the contents of cpu registers and store:

```
{ Execute dca instruction                          }
{ (instruction decoding stage has already set the  }
{ address of the memory location in the mar register)}
{ First, copy contents of acc into mdr register    }
mdr:=acc;
{ Now send signal over bus to memory telling memory }
{ unit to store the contents of mdr register in the }
{ the location specified by the address in mar     }
tomemory;
{ Now clear the acc                                }
acc:=0;
{ Finished execution of dca                        }
```

The control unit will have such micro-programs corresponding to each possible instruction of the computer.

3. The Basic Simulator.

3.1. The simulator and its display.

At this point, it is best to look at the simulator program. Details of how to run this program under UNIX are given elsewhere. Essentially, the simulator expects to read in from a file, the "object" file, a previously prepared and encoded definition of the sequence of PDP-8 instructions that comprise the program to be run. These data are read in; the user is required to specify a display speed appropriate for the terminal in use and to indicate whether the program is to run continuously or is to pause between each stage in the instruction fetch-decode-execute cycle. Once appropriate control parameters have thus been specified, an initial display of the state of the simulated machine is presented. This display is continuously updated as the simulated machine executes the program provided.

The general form of the display is shown below. Bold type has been used to indicate those fields that are high-lighted on the screen (through the "inverse-video" display capability). Fields containing asterisks are filled with specific octal data in real displays. The fields <translated instruction>, <Major stage of instruction> and <minor stage of instruction execution> contain text defining exactly what actions are currently being performed. The field <single step prompt> contains a prompt when the machine is being run in single-step mode and is waiting for a user-response before continuing operations.

```
+-------------------------------------------------------------
|C.P.U.
|
|
|    acc [****] link [*]   pc [****] ir [****]
|
|
|    mar [****] mdr [****] <translated instruction>
|
|
|                                    <Major stage of instruction
|<minor stage of instruction execution>
|_____________________________________________________________
|BUS: control ***** address ****      data ****
|_____________________________________________________________
|MEMORY
|          Address  Contents
|            ****   ****
|            ****   ****
|            ****   ****
|            ****   ****
|            ****   ****
|                 <single step prompt>
+-------------------------------------------------------------
```

There are three components in the display. The top portion of the screen shows data defining the state of the cpu along with the textual descriptions of current activity. The central portion of the screen shows the status of the bus. This portion of the display defines the last data transfer between the cpu and memory specifying whether a memory READ or WRITE operation was performed, the address of the referenced memory location and the value for the data transferred.

Finally, in the third component of the display, there is a "window" into memory.

Even in the basic simulator, the simulated PDP-8 possess more than five hundred "words" of memory. It is obviously impractical to simultaneously display the contents of all memory locations. Instead, the "window" into memory shows that location to which access, for reading or writing, has most recently been made. The two preceding and two subsequent memory locations are also shown. If the most recent memory access was a READ for an instruction fetch, then the memory window will show the sequence of instructions currently being executed. If the last memory access had been a WRITE to a location representing some element of a vector being processed in some loop then the five memory locations shown would represent the array element, the two elements previously processed and the elements still to be processed in the loop. The memory window changes at each memory access.

A typical instantaneous display is shown below. This example represents the state of the machine when it is part way through executing a particular add instruction. This add instruction was at memory location 0201 (all values given are in octal); the program counter has already been incremented and identifies 0202 as the memory location containing the next instruction to be executed. The instruction register contains the value 1207; just below the pc/ir registers the display shows the translation, i.e. *tad 0207* (meaning add the contents of memory location 0207 to the current contents of the accumulator). The acc currently hold the octal value 1, the link is zero.

```
+-------------------------------------------------------------
|C.P.U.
|
|    acc [0001] link [0]   pc  [0202] ir  [1207]
|
|
|    mar [0207] mdr [0002]        tad    0207
|
|
|                                      Executing Instruction
|Data Fetch
|_____________________________________________________________
|BUS: control READ  address 0207      data 0002
|_____________________________________________________________
|MEMORY
|            Address  Contents
|              0205   7402
|              0206   0001
|              0207   0002     0002
|              0210   0003
|              0211   0000
|                       To continue pressRETURN
|
+-------------------------------------------------------------
```

Execution of the instruction has reached the point where the required datum, i.e. the contents of memory location 0207, has been fetched from memory. The text descriptions identify the major stage as being "Executing Instruction" with the minor stage as "Data Fetch". The mar register identifies location 0207 as the last memory element referenced. The bus display shows that the most recent transfer was a READ from memory, at address 0207. The memory window shows locations 205---211 octal. The referenced address is highlighted, and the value read from that address repeated to the right hand side of the column showing the contents of the displayed memory

locations. This value, 0002, is also indicated as being the last data element transferred over the bus, from where it has been copied into the mdr. As values are transferred into, or out from registers, the corresponding display fields are briefly highlighted. Since the program was being run in single-step mode, execution has been suspended at this point awaiting a response from the user.

On receiving a response from the user the program would continue. The contents of the mdr and acc registers (i.e. 0002 and 0001) would be (conceptually) passed to the adding circuit of the ALU of the simulated machine (the ALU is not incorporated in the display). From the ALU, the result 0003 of this addition would be routed into the acc. Execution of this add instruction would then be complete. The simulator would proceed then to the fetch cycle for the next instruction.

The display would change appropriately. The major cycle would be identified as instruction fetch. The contents of the pc, 0202, would be seen to be copied to the mar and a memory read access performed on the appropriate location. The memory display would change to show locations 0200-0204 and the appropriate datum, i.e. the bit pattern representing the next instruction, would be moved from memory to the mdr. Once again, if in single step mode, the simulation would pause for user response.

3.2. An example program, and conventions for arranging programs in the memory of a PDP-8

The program that the machine was executing, when this display recording was made, was:

location	instruction
0200	add the contents of memory location 0206 to the current contents of the acc
0201	add the contents of memory location 0207 to the current contents of the acc
0202	add the contents of memory location 0210 to the current contents of the acc
0203	store the resulting sum in memory location 0211, and clear the acc
0204	reload sum from location 0211
0205	halt
0206	the constant, 1
0207	the constant, 2
0210	the constant, 3
0211	zero, (for the sum)

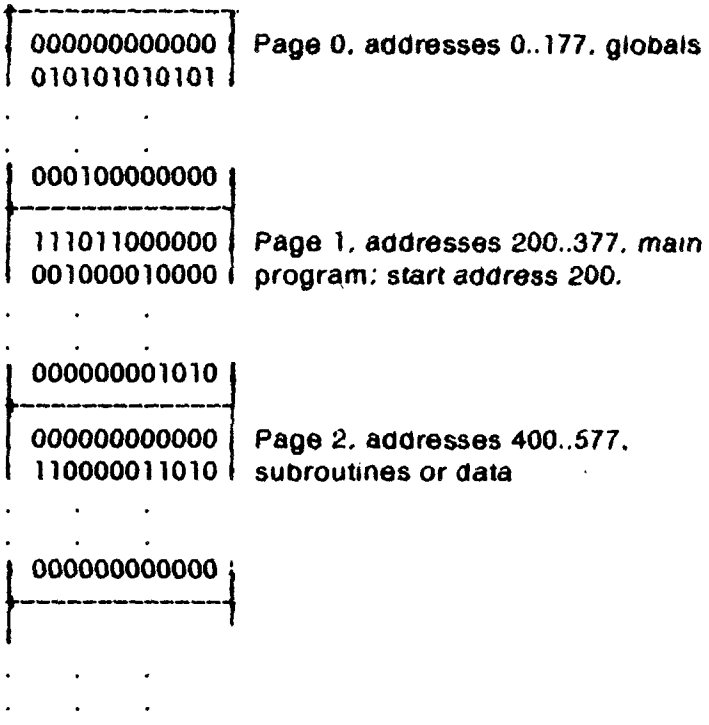
This program fragment introduces some assumptions. Why, for example, the start at location 0200? What did the acc contain before we added in the contents of memory location 0206?

In the general case, deciding where a program is to be executed in memory is a rather onerous task. Typically, a computer will be time-shared, programs for many different users will be in various parts of the machine's main memory with cpu-attention being swapped around between them. If you specify particular memory locations, like 0206 as in the example above, then obviously your program will only execute correctly if your data is loaded into the real memory location 0206. If this location happens to be occupied by some other program then you must wait before you can run your program. Since generally you would want to have programs in all parts of memory you would have to establish some arrangement that particular programs go at particular

addresses. Such an arrangement is hopelessly inconvenient. Consequently, it is normal to try to defer the decision as to where a program is to be run for as long as possible. Usually, one pretends that the program starts at some fixed location such as 0 or 40000000-octal or whatever; all memory addresses are then described relative to this presumed starting point. Then, just before the program is executed (or possibly during execution), these relative addresses are converted to real machine addresses appropriate for wherever it is that the program has actually been stored in memory. This adjustment of addresses in the text of the program can be effected by hardware, or by software or by a combination of both.

Life of course is easier on the PDP-8. Usually such a simple machine is dedicated to a specific application and then it is possible to choose appropriate addresses and incorporate these in the program. There are certain conventions about how memory is utilized. For reasons that hopefully will become clearer later, the first 200 octal (128 decimal) locations, with addresses 0--177, are essentially the place where one stores one's global variables. The next 200 locations, addresses 0200-0377, are where the main program goes along with possibly one or two small subroutines.

Restrictions on how the PDP-8 can access its memory make it appropriate to think in terms of 200-octal (128-decimal) word "pages". Page 0 is for the globals, page 1 for the main program; subsequent pages are for subroutines or data arrays. The basic simulator has four such pages of memory. The simulator has a built in presumption that the main program will commence at location octal-200, i.e. the first word of page 1, and appropriately initializes the pc. It also zeros the acc and link registers prior to executing the PDP-8 program read from file. On a real PDP-8 computer, there were small switches on the "operator's console" that allowed the user to clear registers and appropriately initialize the program counter.



3.3. *Representing a program to be executed.*

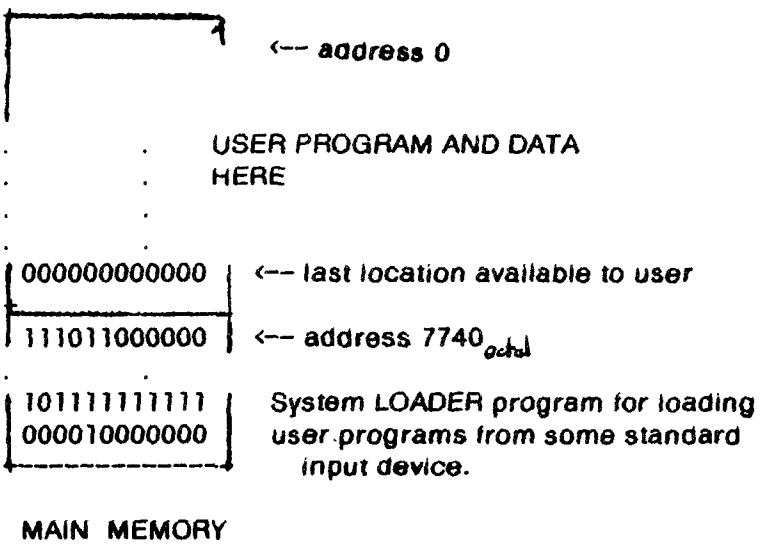
The program, that is to be executed by the simulator, is not of course represented in terms of textual definitions of instructions like *0200 add contents of memory location 0206 to acc*. Instead, the "object" file read in by the simulator contains, as a set of octal numbers, definitions of what memory locations are being used and the instructions, or data constants, that are to go in these memory locations. The object file for the example program given earlier reads as follows:

```
*0200
 1206
 1207
 1210
 3211
 1211
 7402
 0001
 0002
 0003
 0000
$
 0
```

Lines beginning with an asterisk define the first address for a subsequent sequence of instructions; there may be more than one such origin directive in a object file as would, for example, be the case if there were some program text at 0200 and a large pre-initialized data array starting at 0400. Lines beginning with a single space are interpreted by the simulator as specifying data, instructions /constants /whatever, that are to be placed in the next available word of memory. The dollars terminates this section of the "object" file. The simulator contains a couple of routines to read in an appropriately formatted file of octal numbers and store them in the array that represents the memory of the simulated PDP-8.

With a real computer, things tend to be a bit more complex. Firstly, a much more compact encoding would be used for the object file. Here we use a sequence of characters '1','2','0','7' to represent the 12-bit binary pattern '001010000111'. But each such ordinary character takes up eight bits in itself so, really, we have a 32-bit sequence conveying only 12-bits of information. Many more efficient encoding schemes are possible. In other respects, our object file is reasonably realistic; any real object file must convey the same information concerning addresses and their contents.

Of course, unlike our simulator, a real computer can't have built in PASCAL functions for reading in, "loading", the contents of an object file. However, it is possible on almost any machine to write a short "loader" program that can read in object files from some standard device and can fill out the appropriate memory locations with the data representing instructions and constants. Rarely would such an "absolute" loader require more than a score or so of instructions. It is a common expedient to utilize the last few words of available memory to hold this loader program.



Such an "absolute" loader represents about the simplest form of "systems" software on a computer: as we elaborate this "operating system" we find that a rapidly decreasing fraction of the memory of the computer is left for user's own programs.

4. Preparing Programs for the Simulator.

4.1. Assembly language programming with the "smap" assembler.

A program represented as a table of octal numbers may be quite appropriate for a computer but it is barely comprehensible to a human. Of course one can learn to associate a particular binary pattern, e.g. 000-binary 0-octal, with a particular operation that the computer can perform (in this case an "and" instruction). But it is a lot easier if the text of the program had some totally equivalent but more readily comprehensible alternative representation, such as for example the word *"and"* if its an **and** instruction that we want.

It is in fact quite exceptional for anyone ever to have to write out a program for a computer in terms of the binary patterns representing instructions (or their equivalent octal or hexadecimal representations). Instead, one writes in "assembly" language.

There are no general assembly languages, each such language is unique to a particular machine. However, most take essentially the same simple form. Each line of the assembly language program represents information that will eventually correspond to one instruction that can be executed by the computer. Each line is divided up into "fields".

First, there is a label field (familiar to those with some exposure to FORTRAN (with statement "labels"), or even BASIC (with line numbers)). A label can identify a particular program statement and allow reference to be made to it elsewhere as, for example, in a jump (goto) instruction. Labels are also used to identify those memory locations employed for holding constants or program variables. The label field will frequently be left blank (more like FORTRAN than BASIC).

For an instruction, the next "field" will hold, not the opcode itself, but instead some name, or "mnemonic", for the opcode required in the current instruction. Mnemonics are supposedly chosen to remind programmers of exactly what operations a particular instruction entails. **"and"** is a fairly obvious mnemonic for a logical AND instruction; the PDP-8's use of **"tad"** to designate the addition instruction seems odd until one knows that the programmers who thought it up liked to be reminded that they were using two's complement arithmetic (hence, two's complement **add**ition).

What comes after the opcode's mnemonic depends on the particular instruction. If the operation involves for example transfer of data between a cpu register and memory then the next field(s) will contain some specification of the address of the appropriate memory location.

Differences in assembly languages are fairly obvious and closely related to specific machine characteristics. For example, in many ways the NOVA is like a PDP-8 but the NOVA has two accumulators, "A" and "B", where the PDP-8 has but one. On the PDP-8, data can only be stored from the acc register; but on the NOVA data can come from either "A" or "B" accumulators so, in any store instruction, one must specify which register is being referenced. This information may be incorporated in the opcode, e.g. have distinct STOREA and STOREB instructions, or may be specified through some additional instruction field, e.g. STORE (A/B). Machines differ significantly in the range of ways in which the required memory location may be specified, so another area of difference in assembly languages is in those fields wherein the address is defined.

Comments can be included in assembly programs. These are solely for the benefit of the programmer and are thrown away before the program is ever prepared for machine execution. However, given the intrinsic difficulty of reading an assembly language program, comments are essential. (Maybe more essential than the code? The code itself will soon be obsolete, good comments will at least tell the next guy about a program that was once thought worth writing and then maybe he can rewrite

it). Usually, comments are introduced by some special character (typically this character might be '/', ';' or '*'); all text on a line following this comment marker is considered to be comment. Some assembly languages make special provision for comments on each line of code (everything after the 35th character, or other arbitrary limit, gets considered as being a comment).

A program written in assembly language has of course still to be converted into a form that can be processed by a computer. This conversion is the task of an "assembler". An assembler reads the source text of the program and generates from this an object file, similar to those we have already considered, and a listing that details exactly the instructions generated and the storage locations assigned both for those instructions and data. The object file is for input to the loader of the computer that is to execute the code. The listing is used as a reference by the programmer when checking out the actual execution of the program.

The assembly language that one uses is really defined by the assembler program. Different assemblers, all devised to produce code for the same machine, will vary greatly in the facilities that they provide their users. One assembler might allow users to include string constants in their programs, e.g. "Hello World", whereas another assembler might require that the programmers specify each byte of such a string as an octal or hexadecimal constant. (i.e. 110; 145; 154; 154; ...). Such differences in the *assembler* are manifest in the *assembly language* that is defined.

An assembly language program for our "add three numbers" example is as follows:

```
/ Add the values of the three constants in consta, constb,
/ and constc. Store the sum in "sum". Stop with the sum
/ in the acc.
*200
        tad consta
        tad constb
        tad constc
        dca sum
        tad sum
        hlt
consta, 1
constb, 2
constc, 3
sum,    0
$
```

It is more useful to provide comments that attempt to explain the purpose of the subsequent section of code than to append a comment to each individual instruction. Quite often, programmers first working in assembly language will, on being told to include lots of comments, produce a program that reads:

```
*200
        tad a       / add a to accumulator
        tad b       / add b to accumulator
        tad c       / add c to accumulator
        dca d       / store accumulator in d and clear acc
        tad d       / add d to accumulator
        hlt         / halt
a,      1           / a = 1
b,      2           / b = 2
c,      3           / c = 3
d,      0           / d
$
```

While there are indeed many comments in the second version they convey little information.

This fragment of assembly language is written in the format used by the "smap" assembler that has been devised for processing students' PDP-8 programs. For smap, comments should be introduced by '/'. smap accords no particular significance to the column used for instructions, standard tab positions are quite convenient. Following standard PDP-8 assembler notations, label names (which must begin with letters and comprise six of fewer characters) are terminated by a comma ",".

Apart from the lines containing the six instructions, the program has three constants and one variable. For smap, the values of the constants must be given as (unsigned) octal numbers; the variable should be initialized to zero. In more sophisticated assemblers, there are "pseudo-ops" or "assembler directives" that will, for example, allow one to define a decimal constant or create a text string that is to be stored as a sequence of characters in several successive memory locations. smap does not implement any such "data definition" pseudo-ops.

In fact, smap only has two assembler directives. One is represented by the '$' sign in the example. This is used to mark the end of the input so that smap knows that it has read in the complete program. The other assembler-directive implemented in smap is an "origin" directive. An origin directive allows the programmer to specify where a particular bit of code is to go, (presuming of course that one can specify absolute addresses). Here, we want the code to start at 0200. Following standard PDP-8-assembler conventions, an asterisk '*' is used to identify an origin directive. "*200" specifies that the next sequence of instructions go into locations starting at memory location 0200.

The example also illustrates a minor difference in layout between PASCAL programs and typical assembly language programs. In PASCAL, all variables are declared before the code. Usually, though not invariably, assembly language programs have a set of instructions followed by "declarations" of local variables. Some programmers like to gather all variables together and place them subsequent to all code sections, while others will keep each subroutine and its local variables grouped together. Restrictions on how a program may address variables may preclude one or other of these options on a particular machine. On the PDP-8, it will usually be more appropriate to declare a subroutine's local variables immediately after its code segment.

4.1.1. The assembly process.

The operations of an assembler program will be considered in more detail in the lecture course (some details may also be available in an appendix to this document). Essentially, an assembler has to read through the source text of the user's program, take cognizance of origin directives, find instructions and generate appropriate code.

We can consider, briefly, how an assembler might process the example program. First, it can obviously ignore all comment lines, (those beginning with '/'). On finding an origin directive, e.g. *200, it must update whatever record it keeps of where in memory code is to be placed. The first real instruction in the program being assembled reads *"tad consta"*.

The word "tad" can obviously be easily abstracted from this line. An assembler has an internal table, its "symbol table", wherein there are definitions of, in effect, reserved words. The assembler can look up "tad" in this table and confirm that it is a valid instruction mnemonic, that it's a memory reference instruction and so should be followed on the same line by some address specification, and that the opcode that should be written to the object file is "1".

The assembler could then continue to process the same line and would isolate the word "consta". Of course, this is the first time that this word has been encountered. There is no definition for it in the symbol table. From the context, the assembler might infer that its intended to be the name to be accorded to some memory location but it has no way of determining the appropriate address.

There are various ways of resolving problems of such "forward references" (references to variables or program labels whose definitions have yet to be encountered). The simplest solution is to read through the program text twice. On the first pass, the assembler program just finds all variables, and program labels, and determines their appropriate addresses. These data are inserted into the symbol table. Then on the second pass, code can be generated because the assembler program will by then possess the required addresses.

smap is an example of such a two pass assembler. On its first pass, smap will note the origin directive *200. The next line, *"tad consta"*, is recognizable as an instruction; smap can ignore details and simply increment its location counter to 0201. Each additional instruction is processed similarly and results just in an increase in the location counter. When the line, *"consta, 1"*, is encountered the smap assembler will identify a label (that comma makes it very easy to write a label detection routine!). The current value of the location counter is 0206; smap can add the information "name=consta", "symboltype=label", "value=0206" to its symbol table. Similar processing defines the labels "constb", "constc" and "sum" with values 0207, 0210 and 0211.

On its second pass, smap can now process an instruction like *"tad consta"*. The "tad" is recognized as a valid instruction mnemonic; the first three bits of the instruction word being assembled can thus be set to 1-octal 001-binary (the corresponding opcode). "consta" is recognized as a valid operand in an instruction's address field, because it has been defined as a user label. Its value, 0206, can be used to fill out the nine-remaining address bits of the instruction word. Thus, the full instruction "1206" can be generated.

Like any other language, an assembly language has a few rules of syntax. It's not right to say something like *"tad tad"* (because one shouldn't be using "tad" to name a memory location). Similarly, it's wrong to have an instruction such as *"and fred"* if the label "fred" isn't defined anywhere in the program. It's also wrong to use the same label name twice; it may be tempting to use the label "loop," at every point where a loop back is necessary but such usage will just confuse the assembler. One other syntax restriction is the limitation of labels to start with a letter and comprise six, or fewer, characters; another restriction in smap limits the size of numbers (which should not be greater than 4095 decimal).

There appears to be a general agreement among authors of assembler programs that reports of syntax errors in users' code should be as unhelpful as possible. On detecting an error, the usual response of an assembler program is to print a one character message of complaint and to then stop.

smap attempts to be more helpful in that it identifies, reasonably precisely, the point at which the error was detected and the perceived nature of the error. It may well still be necessary to run the smap assembler several times before all errors are eliminated. For example, if smap discovers, during the course of its first pass, that you have defined the label "loop," as being in two different places then it reports this "doubly defined symbol" error (identifying "loop" as the offending symbol along with the point of second occurrence) and then stops. It may also be the case that you have referenced another label, e.g. "loop1" as in "*jmp loop1*", which you have never defined. smap does not detect such "undefined label" errors until it has been able to successfully complete its first pass and is working on the second pass through your program's source text.

4.2. The Instruction Set.

It is at this stage appropriate to consider the main instructions, their opcodes and mnemonics, that are available to you when programming the PDP-8.

The Memory Reference Instructions

There are six instructions for referencing memory, these being **and, tad, dca, isz, jms** and **jmp**. Three of these are used either to transfer data from the accumulator to the main memory (dca) or to combine data from memory with the current contents of the accumulator (tad, and). The jmp and jms instructions provide for unconditional transfers of control and for subroutine calls.

The isz instruction has rather a lot of related uses. What it actually does is first to take the contents of a specified memory location, increment this value by 1, and store the result back into the same memory location; then, the instruction causes the cpu to check whether the addition gave a zero result, i.e. the old value had been -1, if so the program counter would be incremented causing the next instruction to be skipped. There are three common ways in which this instruction is used. It can control simple "for loops"; you initialize some memory location to minus the value of the loop limit and then at the end of the body of the loop, just preceding the jmp instruction that takes you back to the beginning of loop sequence you "isz" this location, when you've been around the loop enough times you will get to skip the jump back.

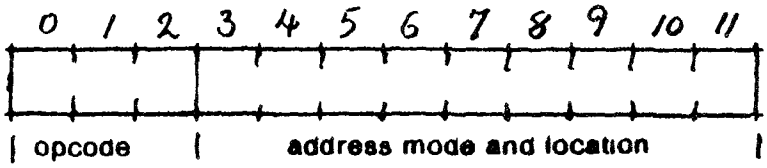
```
        pseudo-PASCAL (with labels)              code

        a:=limit;                          tad limit
        a:=-a;                             cia
        x:=a;                              dca x
100:                            L100,   .
        ( loop body )                       .
           .                              .
           .                              .
           .                              .
        x:=x+1;                 )          .
        if x=0 then goto 101;     )_>---       isz x
        goto 100;                            jmp L100
101:                            L101,   .
```

The other uses both take isz as just a convenient method of incrementing a value, and ignore the bit about skipping on zero. An isz instruction might for example be a convenient way of updating some counter expected to lie only in the range say 10-1000; a single isz instruction can update the value whereas at least three instructions

would be necessary if the current value of the counter had to be loaded into the acc, incremented by 1 and the result stored back. The other, rather similar use, uses the isz instruction to increment an "address pointer". Pointers will be discused further later on; basically they allow reference to some memory location whose address has been derived through some calculation, like for example in code for accessing an array element. If an array is being processed in sequence then quite commonly the isz instruction will be used to increment the address pointer to reference the next element.

Summary of Memory Reference Instructions.



	OPCODE	MNEMONIC
000-binary	0	AND
001-binary	1	TAD
010-binary	2	ISZ
011-binary	3	DCA
100-binary	4	JMS
101-binary	5	JMP

0) **and** : logical AND. The **and** instruction causes a bit-by-bit Boolean AND operation between the contents of accumulator and the data word specified by the instruction. The result is left in the accumulator.

1) **tad** : Two's Complement Addition. **tad** performs addition between the specified data word and the contents of the accumulator leaving the result of the addition in the accumulator. If a carry out of the most significant bit of the accumulator should occur then the link bit is complemented.

2) **isz** : Increment and Skip if Zero. The isz instruction adds a 1 to the referenced data word and then examines the result of the addition. If a zero result occurs, the instruction following the isz is skipped. If the result is not zero, the instruction following the isz is performed. In either case, the result of the addition replaces the original data word in memory.

3) **dca** : Deposit and Clear Accumulator. The **dca** instruction stores the contents of the acc in the referenced location, destroying the original contents of the location. The acc is then set to zero.

4) **jms** : JuMp to Subroutine. (Discussed in next section on addressing modes and subroutines).

5) **jmp** : JuMP. The jump (goto) instruction loads the effective address, calculated during instruction decoding, into the program counter pc.

Operate instructions.

In principle, there are 9 bits available to identify operate class instructions, supposedly therefore allowing some 500 such instructions. It doesn't work that way. Operate instructions are "microcoded". Specific bits are used to designate specific "micro-instructions". One "micro-instruction" clears the acc (i.e. sets it to zero); another clears the link. "Micro-instructions" can be combined in various ways. Thus, one can clear the accumulator and then go on to increment it (so getting the constant 0001 in the acc) all within a single operate instruction. There are however lots of restrictions on the allowed combinations of microinstructions. Consequently, only a few of the 500 odd possible 9-bit binary patterns actually represent valid, executable instructions.

The basic operate "micro-instructions" are:

data manipulation:

i) **cla**, clear the accumulator, i.e. set it to 0000.

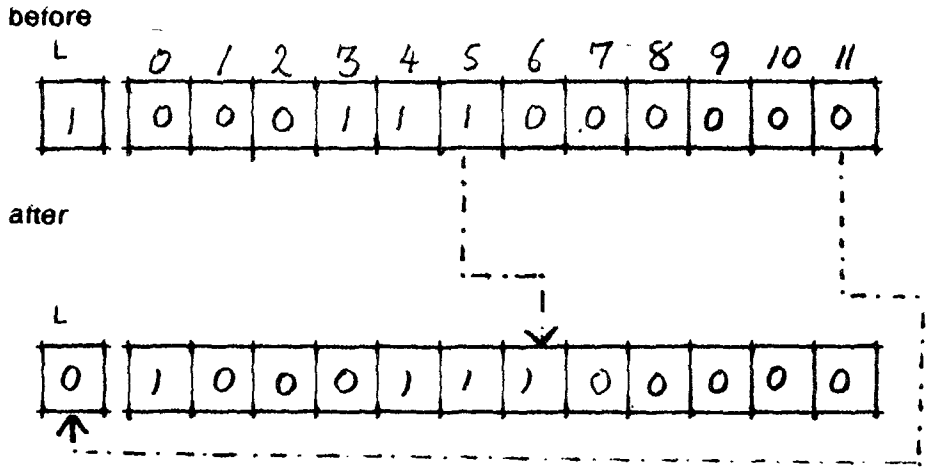
ii) **cll**, clear the link.

iii) **cma**, complement the accumulator, i.e. all binary 1-s become 0-s, all 0-s become 1-s.

iv) **cml**, complement the link.

v) ***rar**, rotate the accumulator and link right. This instruction treats the acc and link as a closed loop and shifts all bits one position right:*

example:



vi) **rtr**, rotate two right, a shift of two places to the right is executed. Both rar and rtr use what is commonly called a circular shift, meaning that any bit rotated off one end of the accumulator will pass into the link and then on again into the other end of the accumulator.

vii) **ral**, rotate left. This instruction treats the acc and link as a closed loop and shifts all bits in the loop to the left performing a circular shift left.

viii) **rtl**, two place rotate left.

ix) **iac**, increment the accumulator, the contents of the acc are increased by 1.

x) **nop**, nooperation is performed; the program control is simply transferred to the next instruction in sequence.

A particularly common combined instruction, which has acquired its own mnemonic, is "**cia**". "cia" combines complementing and incrementing and is the instruction necessary to negate the number in the accumulator.

skips, (conditional jumps over a single instruction)

i) **sma**, Skip on Minus Accumulator. The next instruction is skipped if the contents of the accumulator, interpreted as a two's complement number, is less than zero.

ii) **spa**, Skip on Positive Accumulator. The next instruction is skipped if the accumulator is greater than or equal to zero.

iii) **sza**, Skip on Zero Accumulator. The next instruction is skipped if the accumulator is zero.

iv) **sna**, Skip on Nonzero Accumulator. The next instruction is skipped if the accumulator is non-zero.

v) **snl**, Skip on Nonzero Link. The next instruction is skipped when the link bit is a 1.

vi) **szl**, Skip on Zero Link. The next instruction is skipped when the link bit is a 0.

vii) **skp**, unconditional SKiP. The next instruction is skipped.

viii) **hlt**, HaLT. The computer will stop at the completion of the current instruction.

4.3. A more realistic example program.

Now that we have at least cursorily covered the general concept of assembly language programming, and have considered the instruction repertoire available, its worth looking at a slightly larger program than the "add three numbers" example. All that the program does is loop around, for some fixed number of times, storing into memory a number representing the current count of iterations. The program introduces only one new concept and a further minor detail concerning the arrangement of programs in memory. The new concept is that of pointers and "indirect addressing".

In this program, it's necessary to refer to successive elements of memory as one iterates around a loop. On the first iteration, one has to store 1 in, as it happens, location 217; on the next iteration, a 2 must be stored in location 220; then a 3 in 221 and so forth. Eventually, depending on the number of times the loop must be executed, reference may need to be made to locations 400, 401 etc. Consequently, we would appear to require a "store" instruction in which the address gets changed. It is **not** possible to do this.

Instead, *indirection* is used. A variable is used to record the address of that memory location in which the next number generated is to be stored. After each number is stored, the value in this variable can be incremented so that the new value refers to, or points to the next location to be used. Because it "points" to some required memory location, such a variable is generally known as a "pointer variable" or just a "pointer".

The actual store instruction in the program loop contains the address of this pointer variable. Of course one doesn't want the datum stored in the address specified, that would just cause the value of the pointer to be overwritten. Rather, one wants to indicate that the value of referenced pointer be used to determine where exactly the datum must be stored. Thus, the instruction has to read something like *"deposit the contents of the acc in that memory location whose address is currently stored in this*

pointer variable" or, more concisely, *"dca indirect pointer"*.

This implies that we have some means of indicating that the method of interpreting the address part of an instruction be changed. On the PDP-8, one of the nine address bits in a memory reference instruction is used to designate the "mode" for interpreting the address. There are thus two address modes; the setting of this control bit (which is actually bit 3 of the instruction word) determines how the remaining address bits are to be used. If this bit is zero, we have "direct" addressing. The address given in the instruction is the address in which data is to be stored, or from which data is to be fetched or to which control is to be transferred. If bit 3 is a 1, then "indirect" addressing is necessary. The address given in the instruction designates the memory location of a pointer variable; the required, effective address must be fetched from that location before performing the store, add, jmp or other instruction. (More sophisticated machines typically exhibit a larger number of addressing modes and must use more bits of an instruction word to designate an appropriate mode).

Note that indirection makes an instruction take longer to execute. One extra memory cycle, and some other additional work, are entailed. In decoding the address in an instruction like *"dca indirect pointer"*, the bits that identify the address of the variable "pointer" have first to be abstracted and interpreted so that the appropriate memory location of "pointer" is identified. That first decoding step is, of course, standard to all memory reference instructions. However, if indirection is being used, one must then go on to read from memory the value in this referenced location. So an extra memory access must be performed. It is the value thus read out of memory that represents the real effective address. This effective address must then be used in the subsequent data fetch or data store operation.

The first couple of lines in the program, shown below, initialize a pointer variable, "ptr", so that it contains the address where the first datum is to be stored. Then, a counter is initialized with (minus) the number of times the loop is to be traversed. The body of the loop in this program starts at label "loop" and ends with the "jmp loop" instruction.

The Source Text of the Program.

```
/     This is an example PDP-8 program written more or less in
/ standard DEC PDP-8 assembler style.
/
/     Basically, the program does the following
/
/     for i=1 to 13 do store[i+216]:=i;  (where all numbers are
/           in octal).
/
/     like most PDP-8 programs, it starts at address octal 200.
/
/     what it actually does is,
/
/     1) set a pointer to point to where to store next item
/     2) set a counter to -13 octal (i.e. 7765)
/     3) (ndx, its iteration index, is already zero so does not
/           need to be initialized).
/
```

```
/     loop)
/          4) pick up the value in ndx
/          5) increment it
/          6) store the incremented value back in memory
/          7) pick up the new value of ndx again
/          8) store it "indirect ptr", i.e. ptr is assumed
/               to contain the address where we will store
/               the current contents of the accumulator.
/          9) increment pointer so that it points to the
/               next memory location ready for next time
/               (note, this increment and skip won't cause
/               us to skip since ptr is never going to
/               get to point to location zero).
/         10) increment counter, skip if the value of counter
/               becomes zero. Since we started off setting
/               counter to -13(octal) we should skip after
/               going round the loop the right number of
/               times).
/         11) if haven't skipped this instruction, go back to "loop"
/     12) halt

*20
count, 0
ptr,   0
ndx,   0
*200
   tad tab          / initialize "ptr" from predefined value in "tab"
   dca ptr
   tad x            / and "count" from value in "x"
   dca count
loop,  tad ndx      / increment value in ndx
   iac
   dca ndx
   tad ndx          / store it in next "array element"
   dca i ptr
   isz ptr          / update ptr so it references next array element
   isz count        / increment loop control & check for termination
   jmp loop         / if not yet terminated, go back
   hlt
x, 7765             / equivalent to -13 octal
tab, 0217           / constant specifying where array to start.
$
```

In the body of the loop, the value of "ndx" is incremented and copy of the incremented value stored in the array. The "dca i ptr" is the store instruction that must in effect reference successive elements of memory. The "i" between the opcode and the address is how the use of an "indirect" addressing mode is signalled to the assembler. If a source statement specifying a memory reference instruction contains an "i", between the instruction mnemonic and address label, then smap generates an instruction in which bit 3 is set to 1 (so flagging the indirect mode); usually, bit 3 of a memory reference instruction is zero (so flagging direct addressing).

After the "dca i ptr" instruction, there are two "isz"s. The first simply increments the pointer so that the next location is appropriately referenced prior to another iteration of the loop. The second is the instruction that tests for the end of the loop; when the loop is has been completed sufficient times, the count (which started as a negative value) will reach zero causing the jump back instruction to be skipped. The program terminates at the halt instruction.

The minor detail regarding program layout relates to where on page 0, the "global variables" ptr, ndx and count get placed. The first of these is at *20 rather than *0. It happens that the first sixteen memory locations, addresses 0..17octal, on an PDP-8 typically have rather special uses and should **not** be employed simply to hold global variables.

When this program is processed by smap, the first output produced is a symbol table. This is written to UNIX standard output, i.e. usually the terminal, at the end of pass 1. It defines both the labels used in the program being assembled and also the standard "reserved" words (which are mainly mnemonics for the operate and I/O instructions). A fragment of the symbol table produced for the example program is shown below:

SMAP Pass 1, Symbol table listing:

Symbol table:

Name	Type	Value	Name	Type	Value
adcrb	iot	6601	adsf	iot	6602
and	mri	0000	cia	opr	7041
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
tab	label	0216	tad	mri	1000
tls	iot	6046	tsf	iot	6041
x	label	0215			

On completion of its second pass, smap writes a listing to "standard output" and an object file. Part of the listing follows. It is typical of listings produced by assemblers with various columns of information. The leftmost column identifies the address in which data is to be stored; next follows the value to be stored in that address; the third column, i.e. the rest of the line, shows the source text from which these data were generated. Thus, for instance, the line reading *"0202 1215 tad x"* shows that smap has arranged that memory location 0202 will contain the octal value 1215 which it has determined to be the appropriate binary pattern for an instruction to add the contents of location 0215 to the acc.

SMAP Pass 2 Assembly listing.

```
                    /     this is an example pdp-8 ...
                                    .
                                    .
                                    .
                    /     12) halt
                    *20
0020  0000          count, 0
0021  0000          ptr,   0
0022  0000          ndx,   0
                    *200
0200  1216            tad tab
0201  3021            dca ptr
0202  1215            tad x
0203  3020            dca count
0204  1022          loop,  tad ndx
0205  7001            iac
0206  3022            dca ndx
0207  1022            tad ndx
0210  3421            dca i ptr
0211  2021            isz ptr
0212  2020            isz count
0213  5204            jmp loop
0214  7402            hlt
0215  7765          x, 7765
0216  0217          tab, 0217
```

(The effect of the indirection bit can be seen by comparing the code generated for *"dca count"* (3020 in location 203) and *"dca i ptr"* (3421 in location 210). Bit 3 of the instruction word corresponds to 0400 octal).

smap's other output is the object file. This can be seen to contain the essential summary of the data shown in the first two columns of the listing. Not every address need be specified because, given the starting point for a code segment, most addresses are implicit. Also in this object file, following the code, is a copy of the symbol table. This is purely for the convenience of the simulator system. It allows the <translated instruction> field, in the display, to explicitly refer to the program's original labels. Thus, the display, at the point where instruction "1215" (at 0203) was being executed, would read "tad x" rather than just "tad 215".

The Object File

```
*0020
 0000
 0000
 0000
*0200
 1216
 3021
 1215
 3020
 1022
 7001
 3022
 1022
 3421
 2021
 2020
 5204
 7402
 7765
 0217
$
        45
adcrb    3 6601
adsf     3 6602
and      1 0000
.       . .
.       . .
.       . .
tls      3 6046
tsf      3 6041
x        0 0215
$
```

This example program can be executed on the basic simulator. Since it involves a few iterations around a multi-instruction loop, it takes some time to run with the very detailed display.

The display of program execution is of course transient and provides no permanent record of the program's correct running. The program does change the memory of the simulated PDP-8; the changes effected can show whether or not the program ran successfully. The simulator contains a provision for the contents of memory to be printed off both prior to, and subsequent to, execution of a program.

The printout of memory thus obtained is an instance of a program "dump". It shows addresses and contents of those memory locations that are non-zero. The dump produced by the simulator shows simply the memory contents in octal. Quite commonly, computer systems provide a much more detailed "dump" showing each memory location in several different printing formats e.g. hex, instructions, characters.

The example shown below demonstrates that the program did indeed modify memory by writing in integers 1..13 octal starting at memory location 217.

The Program "dump".

Printout of contents of memory prior to program execution.

```
Address                 Contents

0200 :  1216 3021  1215 3020  1022 7001  3022 1022
0210 :  3421 2021  2020 5204  7402 7765  0217 0000
```

Printout of contents of registers and memory subsequent to program execution.

```
acc : 0000    pc  : 0215    link: 0

Address                 Contents

0020 :  0000 0232  0013 0000  0000 0000  0000 0000
          .    .     .    .     .    .     .    .
0200 :  1216 3021  1215 3020  1022 7001  3022 1022
0210 :  3421 2021  2020 5204  7402 7765  0217 0001
0220 :  0002 0003  0004 0005  0006 0007  0010 0011
0230 :  0012 0013  0000 0000  0000 0000  0000 0000
```

5. The PDP-8's addressing mechanism and subroutine calls.

5.1. Addressing.

In 1965, hardware was relatively expensive so the designers of the PDP-8 skimped on what they provided. The result is a relatively clumsy way of accessing memory. The following description of the addressing mechanism is taken, with minor adaptations, from DEC's documentation.

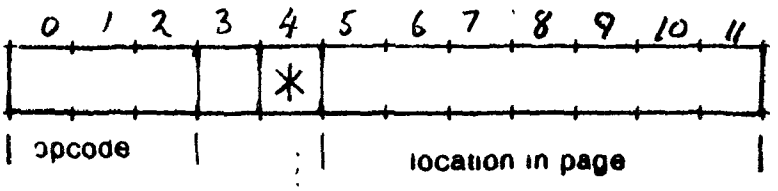
Only nine bits are available to specify a location in a memory referencing instruction. However, a full 12 bits are needed to uniquely address the 4096 locations that are contained in the PDP-8's memory unit.

To make the best use of the available nine bits, the PDP-8 utilizes a logical division of memory in blocks (pages) of 200-octal (128 decimal) locations each as shown in following table (all values in octal):

Page	Memory locations	Page	Memory locations
0	0-177	20	4000-4177
1	200-377	21	4200-4377
2	400-577	22	4400-4577
3	600-777	23	4600-4777
17	3600-3777	37	7600-7777

Since there are 200-octal locations on a page and since seven bits can represent 200-octal different numbers, seven bits (5 through 11) are used to specify the relative location within the page.

The page is specified by bit 4, called the current page or page zero bit. If bit 4 is 0, then the reference is interpreted as being to a location on page zero, i.e. one of the first 200 locations in memory. If bit 4 is a 1, the page address is interpreted to be on the current page, i.e. the page containing the instruction currently being executed. For example, if bits 5 through 11 represent 123-octal and bit 4 is a zero, the location referenced is absolute address 123. However, if bit 4 is a one and the current instruction is in a core memory location whose absolute address is between 4600 and 4777octal then the current page address 123 designates the absolute address 4723.



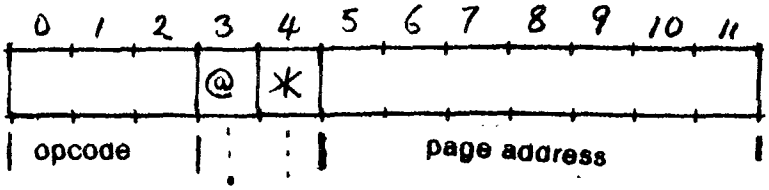
* Current page or page zero bit.

0 => page zero
1 => current page

Indirect addressing.

The scheme described above allowed addressing of 400-octal locations by any instruction --- 200 page zero locations and 200 current page locations. How are the remaining 7400 locations to be accessed?

Bit 3 of a memory addressing instruction identifies the addressing mode. When bit 3 is a zero, the operand is a direct address. When bit 3 is a one, the operand is an indirect address. An indirect address (pointer address) identifies the location that contains the desired address (effective address). To address a location that is not directly addressable, the absolute address of the desired location is stored in one of the 400 directly addressable locations; this pointer address is the used in the memory reference instruction but with bit 3 set to 1. When executing, the machine will fetch the contents of the specified pointer address and then use this to specify the effective address in a subsequent fetch cycle to get the required datum.



* Current page or page zero bit

@ Direct/indirect addressing bit

0 direct addressing. referenced memory location contains required datum.

1 indirect addressing. referenced memory location contains the address of the required datum.

5.2. Subroutine calls.

An effective subroutine call mechanism has to resolve two problems. First, as noted earlier, one needs to maintain some record of where a subroutine was invoked, so that when the subroutine has been completed one can return and resume the main program. Second, one requires some mechanism for passing arguments to a subroutine and retrieving results back when it is complete.

5.2.1. Subroutine linkage on the PDP-8.

The method by which connection is established between a calling routine and a called subroutine is referred to as a subroutine linkage mechanism. A linkage mechanism must provide some means for preserving the address of the instruction to which return must be made. Suppose for instance that we have a subroutine call "*jms sub1*" at location 0204. This instruction will be fetched, the pc incremented, to 0205, and the instruction decoding process carried out. The address 0205 now in the program counter represents the point to which return must be made. This value must be saved somewhere before the program counter is changed to point to the first instruction of the subroutine.

The problem is, of course, where to save the return address. On machines with lots of registers one can use a register; provided that the subroutine knows which register has thus been reserved to hold its return address, then all is well. The PDP-8 doesn't have any registers to spare and so it can't adopt the "linkage register" approach. A more satisfactory approach, using "stack" data structures in the main memory of the computer, will be considered in the lectures. Unfortunately, the addressing mechanisms on the PDP-8 really preclude the stack oriented approach (besides, very few people had thought of such sophisticated tricks back in 1965 when the PDP-8 was designed).

It's necessary on the PDP-8 to use main memory to store the return address. The memory location used must somehow be known to both subroutine and calling program. The only obvious place is the first location of the subroutine. This is of course the address referenced in the JuMp to Subroutine instruction and so it's known to the calling program. A subroutine can be expected to know its own starting address. So, on the PDP-8, the first location of a subroutine doesn't in fact contain an executable instruction; instead its a place for storing the return address.

When a *jms* instruction is executed the effective address must first be evaluated (it may be a direct address or entail indirection). This effective address, held temporarily in the mar register, identifies the start of the subroutine. The current value in the program counter, already incremented and thus pointing to the location following the jms instruction, is written into the memory address specified by the contents of the mar register. The contents of the mar register are then incremented by 1 and this value loaded into the pc. The next instruction fetch will consequently retrieve the first instruction of the subroutine.

The state of the machine at the point where the jms instruction has been fetched, decoded but not executed would be something like:

```
0204   4240        jms sub1          pc [0205] mar [0240]
                                     ir [4240]
0240   0     sub1, 0
0241   1020        tad count

0257   5640        jmp i sub1
```

After execution of this jms instruction one would have:

```
0204   4240        jms sub1          pc [0241] mar [0240]
                                     ir [4240]
0240   0205  sub1, (original "0" now overwritten by return address)
0241   1020        tad count

0257   5640        jmp i sub1
```

with the next instruction fetch from location 241, i.e. the first instruction of the subroutine, and with the return address stored at 0240.

Eventually the subroutine will need to return i.e. it needs to reset the pc to the value 0205 stored at sub1. This return simply requires and "indirect" jump. The execution of the *"jmp i sub1"* instruction at 0257 entails the following sequence of actions in address decoding. First, the bit 4 is isolated, as its a 1 this identifies the address lies on the current page; then bits 5..11 are abstracted and interpreted as identifying the 40th location on the current page, i.e. 0240. Then, because bit 3 is set an indirect cycle is performed; the contents, 0205, of the referenced memory location, 0240, are retrieved into the mar. Finally, the jmp instruction is executed, i.e. contents of mar copied to pc; and thus the return has been made.

It was noted earlier that, usually, the main program is on page 1 (locations 200..377) and that subroutines were on other pages (the basic simulator has only pages 0..3, the advanced version has pages 0..17). So typically, one would be trying to call a subroutine on another page. Note that the following code is erroneous:

```
*200
      jms sub1
      jms sub2
      .
      .
      .
*400
sub1,  0
      .
      .
```

Its not possible to directly reference location 0400 in an instruction at 0200; such a reference is to a different page. One can only directly reference a current page or page zero location. Such a subroutine call must be coded using an indirection via a pointer variable, *viz*:

```
*200
      jms i psub1
      jms i psub2
      .
      .
psub1, sub1
psub2, sub2
*400
sub1,  0
      .
      .
```

The smap assembler should detect and complain about any erroneous cross-page references.

5.2.2. Passing parameters on the PDP-8.

Normally, on any machine, if only a few arguments need be passed to a subroutine then they are passed in registers. Similarly, results are returned in registers. Of course, on the PDP-8, we only have the acc. Consequently, we can only pass one argument into a subroutine via a register and can retrieve only one result back. A single argument and result will be quite sufficient for any example programs that will be written for the simulator.

If there are insufficient registers to pass all necessary arguments then one typically passes to the subroutine the address of some "array" in memory in which additional required arguments are tabulated. This approach will be covered in lectures on "stack-oriented" systems for subroutine linkage and parameter passing.

6. Debugging Assembly Language Programs and the Advanced Simulator.

6.1. The Problem of Errors in Assembly Language Programs.

One soon learns that a successful, error free compilation or assembly represents but a small step towards a working program. On attempted execution the program may run, but produce wrong results, may loop performing forever some mysterious and futile computation or may simply "die" obscurely.

Almost any given algorithm takes far more statements to express in an assembly language than in some high level language such as PASCAL. Even if programmers' error rates, in errors per hundred statements, were constant, assembly language programs would inevitably contain more errors because of their greater length. Typically, programmers find greater opportunity for error in assembly language and their error rate in fact rises, thus compounding the problem.

Errors in assembly language programs tend to be more damaging than those committed in some high level language. Consider for example the sort of error where a loop termination condition is inappropriately expressed resulting in reference to some array element beyond the true array dimension. A PASCAL compiler typically inserts code to verify each computed array subscript. At run-time, such code will trap an erroneous reference and terminate the PASCAL program with an error message indicating the general nature of the error and its approximate location (in terms of a line in the PASCAL source program). Not so with an assembly program. One might conceptually have had an array, equivalent to PASCAL's *"var fft : array[0..127] of integer"*, stored in memory locations 500..677 octal. A run-time reference to the -3rd element of this array would simply be a request for the contents of store location 0475. Such a request is perfectly valid and the contents of that location will be retrieved and used in the computation. Now address 0475 presumably contains either some other data element used in the program or, possibly, an instruction. An instruction, when interpreted as data, is of course just another number (save that it has a perverse tendency to represent the greatest or least element of the array, or whatever else was sought). Whatever it was that was in the referenced location gets used. Consequently, the program will, probably, run to completion but *may* yield erroneous results (depending of course on the particular values in the test data employed).

Even more mysterious behaviours are manifest if, with the same sort of error, one tried to write to the -3rd element of the array. In so doing, one would *overwrite* the original and correct contents of memory location 0475. The processing of the array might well be completed satisfactorily, so enhancing the programmer's confidence in that processing routine and diverting the search for the error from its true location. Such an error of overwriting will only be apparent if, at some later stage of the computation, the original instruction or datum is again required. If the overwritten location was used to hold an element of program data then the new value, erroneously written into the location, will cause unexpected, and inexplicable results to be obtained elsewhere. Most bit patterns representing numbers can also be interpreted as instructions. Consequently, if the overwritten location was supposed to contain an instruction then there will be an attempt to interpret the new erroneous value as an instruction. The attempt may be successful and some instruction, albeit not the intended one, will get executed.

The most common type of assembly language error, apart from the kind of erroneous data addressing just discussed, probably relates to arbitrary transfers of control. Jmp, jms and skp instructions all admit any degree of misuse. In PASCAL programs are forced to be well structured. It is, for example, impossible to jump into the middle of a "for-loop" and thus end up checking against loop limits that have never been initialized. Such uncontrolled transfers are trivially realized at assembly language level. Although all programmers intend to write clear well structured assembly programs from algorithms expressed in some pseudo-PASCAL, the liberty allowed at the

assembly language level frequently subverts these intentions. Programs with complex control flows are created which, after a couple of rounds of modification become completely incomprehensible and in which all transfers involving any address computation become unreliable.

One particular cause of errors, relative jumps defined in the program source text, is eliminated with the smap assembler. Quite often, one wants to code an instruction that will effect a jump around say the following five instructions (as for instance when encoding the false branch of an "if" statement). One may not feel inclined to invent an additional label on the instruction to be jumped to; rather one simply wants to say "jump over five instructions". Many assemblers do provide a notation, e.g. "jmp .+6", which achieves the desired effect. Unfortunately, the subsequent addition of a couple of extra instructions to the "true" portion of the conditional may then have overlooked side effects. (The lack of such relative addressing in smap is due solely to the laziness of the implementor and not to any overt intent to prevent such self-inflicted errors).

Other common, but trivial errors, include specifying the wrong address mode (so perhaps resulting in the use of the address of a variable rather than its value), or specifying the wrong instruction. In their normal helpful manner, designers of assemblers contrive to provide pitfalls such as instructions with very similar mnemonics. It is difficult, especially when working with a listing produced on a low quality printer, to notice that one has written lb where lh was intended. The program that loads one byte, when two were desired, runs but does not achieve its intended purpose. Not surprisingly, paranoid delusions concerning malevolent machines are endemic amongst those first learning assembly language.

The normal response to mysterious bugs in a PASCAL program is to throw in a whole series of "write" statements. These extra "tracing" statements allow one to print off tables containing the values of all important control variables, or to print appropriate messages at procedure invocation, loop termination etc. With a few well chosen trace statements, a programmer can usually rapidly localize the erroneous portion of code.

This technique is less readily applied to assembly language programs. A one line PASCAL trace statement, e.g. *"writeln('entered sort routine, n = ',n:8);"*, may well require a score, or more, of assembly language instructions. A "few well chosen trace statements" may consequently represent more lines of code than the original erroneous program. Even if one had the tenacity to insert the additional tracing code (and sheer luck sufficient to perform the insertion without introducing additional errors) one has a further problem. The new code will have resulted in significant changes to the detailed construction of the program. The address containing the instruction overwritten in the original buggy program, or address to which erroneous transfer was made, now most probably contains something different. One still has a bug to chase, but its a different bug from that in the original program and may express itself in some completely different fashion.

The response of despair is to use a "program dump". Earlier we examined the "dump" produced from the basic simulator; it showed the original memory contents and the contents on completion of the program. If the program contains errors one can get a dump made at the point at which it died, or was aborted if it was stuck in some loop. This dump represents the state of memory at some unknown time after the commission of an error. One may examine the contents of memory as presented in this dump. One searches for evidence of deviations from the required behaviour of the program, such as memory locations that contain unexpected and inappropriate data. From such evidence one may be able to identify the source of the disruptions. Debugging from program dumps requires considerable patience, care and an appropriate mental attitude. It is a task apparently well suited to acolytes of the Church of Latter

Day Saints.

6.2. Interactive Debugging.

The best approach to debugging assembly language programs (and high level language programs for that matter) is to work interactively, at a terminal, controlling the execution of the program. Single-stepping of a program is a powerful technique. One does not work one-cycle at a time (as in the basic simulator), instead one executes one statement (i.e. one instruction in assembler) at a time and views the changes effected before proceeding.

The problem with single-stepping is that its only viable when one has brought execution of the program very close to the point where the error is actually committed. Suppose for example that one had a binary search routine in which the error related to the test for the two array pointers passing one another. Even in the simplest contrived test case, it might require several hundred, or several thousand, instructions to be executed before the erroneous code was reached. It is impractical to single step through several hundred instructions in order to reach the region of an error.

A more flexible system is required wherein one can run the program normally up to some point chosen to lie just before the suspect code. Such a point, whereat execution is to be temporarily suspended, is referred to as a "breakpoint". On reaching a breakpoint, control should be transferred to some interactive routine that will let the user inspect the values of chosen variables, set subsequent breakpoints, and, if necessary, to invoke single-stepping of the next few instructions. When the user is satisfied, execution of the suspended program should be resumed.

It is easy to implement such a system on a simulator. One simply provides an extra array of boolean variables of the same size as the array representing memory. Instructions at which there are to be breakpoints are flagged by setting the corresponding boolean to true. The simulator program can check in this boolean array to see if a breakpoint is needed prior to the next instruction fetch. If a breakpoint has been reached, the simulator can call a "break" handling routine that provides the user with various options for inspecting memory and cpu registers. When the user indicates that the program is to continue, the break routine is exited and simulator resumes with the next instruction fetch. It is possible, but considerably more difficult to implement a similar interactive debugging function for a real computer.

The "advanced simulator" contains some break (debugging) functions loosely modelled on standards such as DEC's DDT and UNIX's adb. As well as allowing for control over the execution of the program, the break functions enable the user to define the required detail in the display of the simulated computer. Control is passed to these break functions prior to attempted execution of the user's program; this allows the initial setting of display options and, if necessary, initial breakpoints. The simulator attempts to trap all errors, such as erroneous memory references, and pass control to the break functions.

6.2.1. The "break" functions.

The break functions announce themselves by clearing the screen, printing the current value of the program counter (Break address : xxxx) and then prompting the user with the prompt "break>". The user may then enter a variety of commands.

1) +/- commands. These turn on (+) or turn off (-) various run-time displays. These display options are considered in the next section.

2) : commands. These commands enable the user to set, or to remove breakpoints, resume execution of the suspended program, abandon execution entirely and to obtain various other information.

3) Memory display commands. These allow the contents of specified locations in the memory to be displayed in various chosen formats. One can for example view a particular range of locations on the assumption that these locations should contain instructions; comparison of the instructions displayed with those in the original program listing will reveal any that have been overwritten with program data. The basic format for these commands is "<address>[,<repeat-factor>]/<format>"; that is, an address (either an octal constant or the name of one of the program labels) optionally followed by a repeat-factor (given as a comma followed by an octal number), a "slash" character and then a one character format specification.

Each command entered by the user is processed immediately; on completion of a command, the break function again prints the "break>" prompt. This cycle only terminates when the user enters ":c" (for "continue" i.e. resume/start execution of user program) or ":q" (for "quit" i.e. abandon it all).

The full set of ":" commands comprises:

:b set breakpoint.
:c continue execution of PDP-8 program.
:d "dump" registers and memory to terminal screen.
:q terminate PDP-8 program.
:r "dump" registers to terminal screen.
:s list user defined symbols (labels).
:u undo, i.e. remove, a breakpoint

For the :b and :u commands (set and remove breakpoint), the break function responds with "Address for breakpoint>". An address must be either an octal constant or one of the program labels. Breakpoints can only meaningfully be set at locations containing instructions that are fetched. Breakpoints set on data elements, or on the first word of subroutines, are ineffective for no instruction fetch is performed on such locations.

The :r command simply prints the current contents of the acc, link and pc registers. These values are also printed for the :d command, which then proceeds with a memory dump similar to those previously illustrated and discussed. The :s command is simply a convenient way of getting a summary of some of the data available in the symbol table listing.

The formats in which memory words can be displayed are:

c one ASCII character/word.
d word value in signed decimal.
i word interpreted as an instruction.
o word value in octal.
p two 6-bit characters packed in a word.

Some example memory display commands might be (user input shown in bold type):

```
break>
loop,10/i
loop   0204 / tad  ndx          0205 / iac
       0206 / dca  ndx          0207 / tad  ndx
       0210 / dca i ptr         0211 / isz  ptr
       0212 / isz  count        0213 / jmp  loop
```

(Here the user requested that the contents of 10 (10 octal, i.e. 8 decimal) memory locations, starting at the address of the label called loop, be displayed as if they represented instructions).

```
break>
204.4/o
loop   0204 /  1022        0205 /  7001
     0206 /  3022        0207 /  1022
```

(Here, the request was for the contents of four memory locations to be displayed in octal).

```
break>
215/d
x      0215 /   -11
```

(This was an example of print out as a signed decimal number).

6.2.2. The displays in the advanced simulator.

The display options are set using the "+" and "-" commands in response to the "break>" prompt. The commands are:

- * list current display settings.
- c set/reset CPU display.
- d set/reset memory display for "data".
- i set/reset memory display for "instructions".
- l set/reset logging of registers on interrupts.
- p set/reset display for status of peripherals.
- s set/reset single step mode.
- t set/reset instruction translation display.

The initial, default, settings for the various options can be obtained by "+*".

```
break>
+*
Current display settings:
C.P.U.                        :      true
"Data" Memory Window          :      true
"Instructions" Memory Window  :      true
Status of Peripherals shown   :      true
Single step mode              :     false
Translation of instructions   :      true
```

Some example commands are:

```
-d
```

(This command would turn off the "Data" Memory Window during any subsequent display of program execution).

```
+s
```

(This command would cause subsequent program execution to be in single step mode, i.e. the simulator would pause between each complete instruction until a user response is received).

Even at their most elaborate, the displays in the advanced simulator are less complete than those of the basic simulator. The C.P.U. display takes the following form:

C.P.U.

acc [****] link [*] pc [****]

Time : **** x 10^-7 seconds

Only the acc, link and pc registers are displayed. There is a new component, replacing mar mdr and text fields; this is a crude time estimate. The estimate is based on a one micro-second memory cycle time and various somewhat arbitrary measures of other components in instruction times. (Rather than use fractional micro-second times, the units are 10^-7 seconds). If the CPU display is on, then the register displays are updated each time the corresponding register is changed; the time is updated at the end of each instruction. If the instruction translation option is set then, when an instruction has been completely decoded its "disassembled" form is displayed (it appears to the right of the point where the pc register would be displayed, irrespective of whether or not the CPU display option is enabled).

There are two separate windows into memory. One shows the region last accessed on an instruction fetch, the other the region for the last data fetch or write operation. These display components are separately selectable. Individually they are like the single display of the basic simulator. The memory displays, if selected, appear on that region of the terminal screen below where the CPU display would go. An example with both memory windows selected is:

MEMORY

Instructions		Data	
Address	Contents	Address	Contents
0176	0000	0214	7402
0177	0000	0215	7765
0200	1216	0216	0217
0201	3021	0217	0000
0202	1215	0220	0000

The peripherals display, which occupies the bottom portion of the screen, will contain messages identifying any peripherals which currently have "flags" set, (and what data is in any related input buffers). The status of the interrupt line is also reported.

Maintaining an elaborate display considerably slows down the simulation. In normal use, the display may be limited to just the translated instruction while the program is executing code that seems correct. More elaborate options can be selected when the program has been run up to a breakpoint just preceding some, as yet unidentified, error.

6.2.3. A worked example of the use of the break package.

Systems like interactive debugging functions really require live demonstration on a program containing genuine bugs. Contrived examples rarely suffice, for the bugs are known and easy to discover. This example is only partially worked out; it is intended to suggest some uses for the debug functions.

The program is a slightly modified variant of that presented earlier when discussing indirection. The intended change was simply that the numbers 1..13 octal be stored on page 0, in locations 150..162, rather than in locations following the code.

Another change, introducing the bug, has also been made.

If the program is executed as is, it will loop seemingly endlessly. It will also be seen to be executing "and" instructions, although there are none of these in the text.

```
/ a buggy version of the standard example program for
/ storing some data.
*20
count, 0
ptr,   0
ndx,   0
*200
   tad tab
   dca ptr
   tad x
   dca count
loop,  tad ndx
   iac
   dca ndx
   tad ndx
   dca i ptr
   isz count
   isz ptr
   jmp loop
   hlt
x, 7765
tab, 0150
$
```

Following such an unsuccessful attempt at execution, the program was rerun with a breakpoint at "loop". The recording shows the initial settings of the controls, only the instruction translation display was desired so the others were turned off.

```
BREAKPOINT
Break address    : 0200
break>
-c
break>
-d
break>
-i
break>
-p
break>
:b
Address for breakpoint>
loop
break>
:c
```

Each time the program completed a cycle it stopped at label loop. It would then have been possible to inspect the values in "count", "ndx" and "ptr", or in any other memory locations. Instead, the program was just "continued", i.e. :c response to break>. This process was repeated until it was noticed, through the translated instruction display,

that mysterious "and" instructions were being executed. Then, at the next breakpoint, various displays of memory were requested.

```
Break address    : loop
break>
:c
  .
  .
  .
 ("and" instructions observed in translated instruction display)
Break address    : loop
break>
:d

acc : 0000   pc  : 0204   link: 0


Address                  Contents

0020 :  0025  0207  0000  0000  0000  0000  0000  0000
      .     .     .     .     .     .     .     .
0150 :  0001  0002  0003  0004  0005  0006  0007  0010
0160 :  0011  0012  0014  0015  0016  0017  0020  0021
0170 :  0022  0023  0024  0025  0026  0027  0030  0031
0200 :  0032  0033  0034  0035  0036  0001  0000  1022
0210 :  3421  2020  2021  5204  7402  7765  0150  0000
break>
200.20/i
      0200 / and  0032          0201 / and  0033
      0202 / and  0034          0203 / and  0035
loop  0204 / and  0036          0205 / and  0001
      0206 / and  0000          0207 / tad  ndx
      0210 / dca i ptr          0211 / isz  count
      0212 / isz  ptr           0213 / jmp  loop
      0214 / hlt                0215 / opr 204
tab   0216 / and  0150      x   0217 / and  0000

break>
:q
Program finished.
Do you want final contents of memory and registers "dumped" to a file?( Y or N)n
```

These displays showed that some anomaly had occurred at the point where the iterative cycle should have terminated. The value 13 should have been the last value stored in memory, at location 162; after storing this value, the program should have halted. Instead, this value is missing and the program has continued its cycle storing values from 14...36 in successive memory locations. It has then again changed and stored 1 and 0 in locations 205 and 206. Because it did not stop as expected, it has overwritten part of the program with the last few values generated. These values, e.g. 0036, when interpreted as instructions are perceived as "and" instructions, 0036 = and 036.

The example was again rerun. This final time, when the loop was near to termination, count = -2, the displays of instructions and data fetches/writes were reenabled. The last few instructions preceding the expected loop termination were

executed in single step mode. Through these methods, the source of the error was identified.

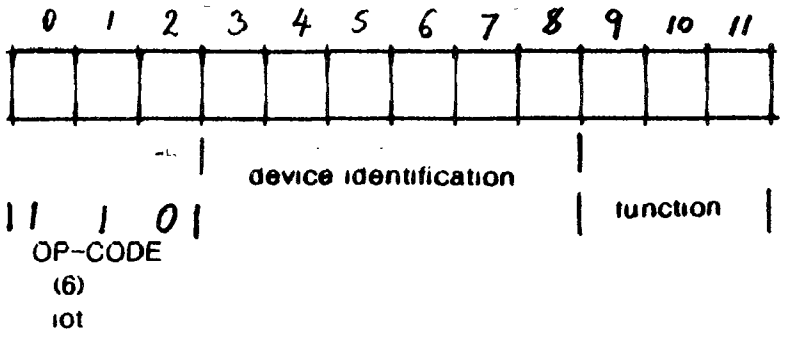
7. Talking to peripheral devices.

So far, our example programs have been contrived to be wholely self contained. All data that they have required were defined internally. The only "results" generated are the displays of successful execution (possibly evidenced by the final "dump"). More typically, programs are intended to process data defined at run-time and need to be able to perform both input and output.

The most common requirement is that a program be able to read a sequence of characters typed in on a keyboard and to write some appropriately transformed character sequence to a teletype (i.e. printing terminal) or a video screen. The advanced simulator incorporates a "teletype" and "keyboard" which can be operated through the standard I/O instructions of the PDP-8.

When a key is struck on a keyboard, electronic or electromechanical devices cause a sequence of voltage pulses to be transmitted over connecting wires to a keyboard control unit. The particular sequence of voltage pulses encodes the required character in some agreed manner. The controller contains an eight-bit "buffer" register in which it builds up the character as it receives the successive voltage pulses. Similarly, a teletype controller contains a buffer register in which it stores the code for the character to be printed or displayed on the terminal. The circuitry in the teletype controller reads the successive bits out of this buffer register and uses them to generate voltage pulses for controlling the electronic or electromechanical display device.

These "buffer" registers in the device controllers are also indirectly accessible to the cpu. The cpu can send, via the bus, a signal to, say, the keyboard controller asking it for the current contents of the keyboard buffer register. This datum would be returned, via the bus and mdr register, to the acc where it could be analyzed by the program. Similarly, the cpu can send a signal to the teletype controller telling it to copy 8-bits from the acc (sent via mdr and bus) into the teletype (tty) buffer register, and then to use these data to generate appropriate signals as will cause the corresponding character to be printed.



On the PDP-8, such requests to device controllers are effected through *iot* (input-output) instructions. The opcode for this instruction is, as always, in bits 0-1-2; for iot's, the opcode is 6. The remaining nine bits of the instruction word identify the device controller, to which a command is to be sent, and the signal identifying the function that the controller is required to perform. Six bits are used to identify the device, the last three bits encode the function.

For example, device identifier 04 happens to designate the teletype controller. A command sent to this tty controller specifying function 4 will cause that controller to take a copy of the low order 8-bits from the acc (via mdr and bus) and transmit these to the connected terminal device. The full instruction word would thus be "6044". There is a mnemonic for this instruction, *tpc* (Teletype Print Character).

Messages could thus apparently be printed on a teletype by means of the

following code:

```
/ initialize a pointer to the address of the start of a message.
/ message to consist of individual ASCII characters
/ one per word and terminated by a zero word.
        tad amsg
        dca ptr
/ loop.
/       cycle around until the zero word marking end of message
/       is found; transmit each character to tty.
loop,   tad i ptr
        sna
        jmp  done       / found end marker, leave this section
/ have character, transmit to tty.
        tpc
/ increment ptr so that it points to next word, then go
/ back to continue loop
        isz ptr
        jmp loop
amsg,   msg
msg,    110     / H
        145     / E
        .
        .
        0       / end marker
```

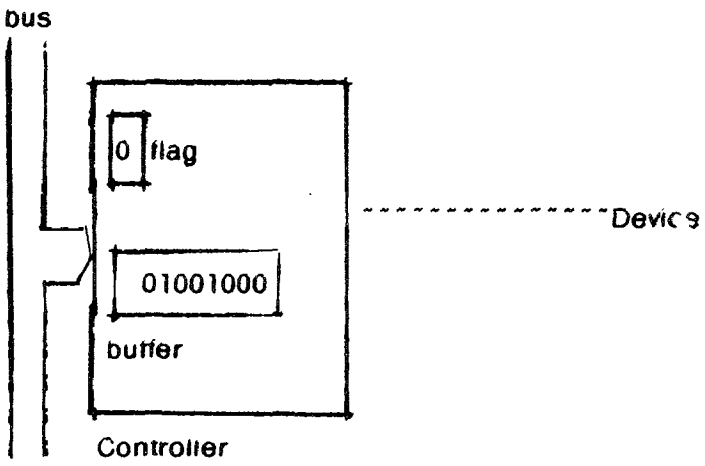
Though reasonably logical, such code would not in fact work correctly. The most likely result of executing the code segment would be for the teletype to try to produce a "rubout" character.

The cpu, the bus, the controllers all run in microsecond time quanta. The cpu sends data to the tty controller --- and they're there, in the teletype buffer register, in a couple of microseconds. Then its up to the controller to forward these data bits to the device.

Electromechanical devices proceed at a slower pace. A printing terminal may take one tenth of one second to print a character; even a video terminal takes of order one one-hundredth of a second. The physical nature of the devices limits the speed at which they can accept data; the controller must limit its transmission rate to what the device is capable of handling. Consequently, a teletype controller will probably have to hold, in its buffer register, its copy of the bits representing a character for something like one tenth of a second. If, as in the example program, the cpu attempts to send a second character, only a few microseconds after the first, then these new data bits are "or"-ed in with those representing the previous character. The entire character sequence representing the message will be loaded into the teletype buffer register before the controller would have completed sending of the first one bit of the first of these characters.

If I/O is to proceed correctly, the cpu must wait until a device controller has completed one task before it gives it another. Its simple for the circuits in the device controller to register completion of a task; in the example of the teletype controller the task would be complete when the last bit of the current character had been successfully transmitted. At that point, the controller could set a one-bit boolean variable, or flag, to indicate that it was ready to process more data. So, as well as a buffer register to hold data being transmitted or received, a typical controller will incorporate a "flag"

register.



Like the buffer register, a controller's flag register can also be accessed by the cpu. What one normally wants to do is make a conditional jump, or skip, if the flag is set so indicating that the device is ready. A typical I/O instruction testing a flag is "6041" *tsf* (Teletype Skip Flag): if the tty controller's flag is set then execution of the tsf instruction causes the pc to be incremented by one so that the next instruction in sequence is skipped.

Through the use of such instructions one can create loops that wait until a device is ready:

```
twait,  tsf
        jmp twait
```

The program will loop around this pair of instructions until the teletype controller sets its flag indicating that it can accept another character.

We now have almost the complete mechanism for a viable "send message" routine.

```
/ loop,
/       cycle around until the zero word marking end of message
/       is found; transmit each character to tty.
loop,   tad i ptr
        sna
        jmp  done       / found end marker, leave this section
/ have character, transmit to tty.
        tpc
/ wait until it got there!
twait,  tsf
        jmp twait
/ increment ptr so that it points to next word, then go
/ back to continue loop
        isz ptr
        jmp loop
```

This program sends the first character of the message and waits appropriately until that character has been successfully transmitted. The program then loops back, collects the next character and sends it. However, the "ready" flag on the teletype controller is still set as a consequence of the successful transmission of the first

character; so, the wait loop is immediately satisfied and the program proceeds at once to sending 3rd, 4th and subsequent characters. It is necessary to clear the flag at some appropriate point in the loop (extra redundant clear flag operations are not in any way harmful). There is another iot instruction, 6042 *tcf* (Teletype Clear Flag), which performs the clear operation (on the simulator it also clears the data buffer).

A correct version of the program is:

```
loop,  tad i ptr
       sna
       jmp  done      / found end marker, leave this section
/ have character, transmit to tty.
/ but first clear ready flag if set
       tcf
       tpc
/ wait until it got there!
twait, tsf
       jmp twait
/ increment ptr so that it points to next word, then go
/ back to continue loop
       isz ptr
       jmp loop
```

The two instructions 6042 (tcf) and 6044 (tpc) can, and normally would be combined into a single 6046 (*tls*) instruction.

Reading data from a keyboard requires a similar loop. One must wait testing the flag on the keyboard controller until it gets set to indicate that its ready to give new data to the cpu. The two instructions most needed to control the keyboard are *ksf* (Keyboard Skip Flag) and *krb* (Keyboard Read Buffer); the krb instruction clears the flag ready for next time. Thus, an appropriate wait loop for getting data from a keyboard might be:

```
kwait, ksf
       jmp kwait
       krb
```

The real computer terminal is of course used to control the simulator. It was considered that any arrangement whereby it was also used as the terminal on the simulated PDP-8 would prove too complex (one would need to indicate whether the next character typed was intended to control the simulator or constituted data to be read by the PDP-8). Instead, the simulated PDP-8 has a pseudo-keyboard and pseudo-teletype that are, in fact, standard UNIX files. Data are read from and printed to these files, one character at a time, using standard PDP-8 iot instructions. (Currently, the files must appear in the user's working directory with fixed names; the input file being ".8.kbd.1").

The following is an example program for the simulator that uses these input and output mechanisms. It copies characters from the pseudo-keyboard to the pseudo-teletype; the program terminates after copying the first zero, '0', character found in the input file.

```
/ Copy characters from keyboard to teletype.
/ Stop after first zero character.
*200
        cla
loop,   ksf             / wait for keyed input
        jmp loop
        krb             / read it
        tls             / print it
wait,   tsf             / wait for printing
        jmp wait
        cla             / check if it was the character zero,
        tad const       / i.e. 60 octal.
        sza
        jmp loop
        hlt
const,  60
$
```

The execution of the two instructions *tsf; jmp wait* would, on a real machine, take about 5 microseconds. If one were waiting one tenth of one second, as one would if waiting for a teletype to print a single character, then one would have to go around that loop something like twenty thousand times. On the simulator, the devices have been speeded up by a factor of some three orders of magnitude. Wait loops are still noticeable, but one only has to go around some twenty times, rather than twenty thousand.

8. More advanced I/O --- interrupts.

The concept of interrupt driven I/O is elaborated in the lecture course. The main example used concerns an PDP-8 system on which data is to be acquired, at fixed time intervals from an analog to digital converter, simultaneous with the processing of previously obtained data. Acquired data have to be buffered in memory until they can be processed.

The example program used is included here for reference. The "clock" and "a/d" converter incorporated in the simulator do not correspond to any real DEC manufactured devices and their device identification numbers and function codes are arbitrary.

```
/
/ Simple example of an interrupt driven program.
/
/     It simulates a simple kind of laboratory data acquisition task:
/     (n.b. its not a perfect simulation)
/
/     every 200microseconds sample the d/a, store data in a buffer while
/     processing previously acquired data
/
/     We assume that time taken to process each datum is variable
/ and that, although processor can in long run keep up with data
/ acquisition, there will be short term fluctuations where data acquisition
/ gets ahead. So we have a circular buffer (512 words long starting at
/ location 512) with two pointers into it --- one for the functions
/ that process data and one for those that put the data in. We assume
/ that acquisition process will never get 512 words ahead of processing
/ so that there is no need to check for buffer being full. We assume
/ that it is sufficient just to compare pointers, if they are equal
/ that means that processing has caught up with acquisition and must
/ wait, otherwise assume it means some data available for processing.
/
/     the clock interrupts every 200 microseconds (approx) once started
/
/     the a/d can be started, it stabilizes and can be read after
/     time equivalent to about 50 microseconds
/
/     we use both under interrupt control
/
/    algorithm:
/     initialize
/     start clock
/     until there is data to process do loop;
/
/    processing data:
/     pick up next unprocessed datum from circular buffer, analyze it
/          (the analysis routine is a phony, we just shuffle bits
/          around, number of iterations depends on number of binary
/          ones in the datum (which is actually a random number
/          generated by the program that simulates all this))
/
/
/
/     interrupt analysis:
/          save system
/          skip chain to find who done it
/
/          if clock, clear clock flag (leave it running so will
/               get another interrupt in 200time units)
/             and start an a/d transfer
/
/          return from interrupt
/
/          if a/d, clear a/d flag, read value and save it in
/               buffer
```

```
/
/           return from interrupt
/
/
/
/ the main program is one page 1, locations 200-240 octal
/
/ the data analysis routine is on page 3, locations 600-630 or so
/
/ page 0 has a few globals etc, and also the hardwired interrupt
/     entry point at location zero
/
/ the interrupt analysis, skip chain etc is on page 2 around locations
/     400 etc.
/
*0
/save program counter on interrupt
        0
/go off and identify cause of interruption
        jmp i pints
pints,  0400
/ places to save acc and link
accsav, 0
lnksav,0
*20
ptr1,   0
ptr2,   0
mask,   1777
bstart, 1000
datum,  0
*200
/ main program starts here
start,  cla cll
/ initialize both pointers to start of buffer
        tad bstart
        dca ptr1
        tad bstart
        dca ptr2
/ start the clock,
        6504
/ turn interrupts on
        6001
/ (a rather unsafe check to see if data waiting)
loop,   tad ptr1
/ compare pointers to see if more filled in
        cia
        tad ptr2
        sna cla
        jmp loop
/ since some data, get it and save in datum
        tad i ptr1
        dca datum
/ update pointers, remember its a circular buffer so this
/ is a bit fussy
```

```
        tad ptr1
        iac
        and mask
        sna
        jmp repos
        dca ptr1
        jmp proc
/ to repos, we ran off the top of the circular buffer so reset
/ pointer back to bottom
repos,  tad bstart
        dca ptr1
/
/ to proc, call the subroutine that actually processes datum
proc,   jms i procs
        jmp loop
procs,  0600
/
/ Page 2,
/       first save the acc
/       then save link
/
/       then go down skip chain trying to find who interrupted
*400
ints,   dca accsav
          rar
          dca lnksav
/       first check the clock, device 50, operation 2, skip
/       if clock has raised its flag
          6502
          skp
/       ok, it was clock, go do something about it
          jmp clksrv
/       otherwise, probably the a/d
/       device 60, operation 2, skip if a/d stablized and flag set
          6602
          skp
          jmp ad
/
/ oops, something interrupted but don't know what,
/       best stop dead
          hlt
/
/
/ here is code for controlling return from interrupt
/       we restore the link
xit,    cla cll
          tad lnksav
          ral
/       and restore the accumulator
          tad accsav
/       and turn interrupts back on, note that
/       this is actually delayed a couple of instructions to
/       give us a chance to get back before another interrupt
/       could be accepted
```

```
        6001
/       How to return? Just go back to whereever address stored
/       in location zero says
/
        jmp i 0
/
/
/       serve the clock.
/               that means just clear its done flag, leave it running
clksrv, 6501
/
/       and it also means that we should start the a/d on its next sample
        6604
/       but that is all so return from interrupt after
/       restoring status appropriately
        jmp xit
/
/
/       an interrupt from the a/d, it means the next sample is ready
/       read it into the accumulator, then store it away,
/       update pointer into buffer (usual fuss for a circular pointer)
ad,     6601
/       value from a/d, 10bits, now in acc
        dca i ptr2
/       value now saved, do the pointer updating
        tad ptr2
        iac
        and mask
        sna
        jmp reset
/       haven't hit end of circular buffer so just
/       carry on
        dca ptr2
/       go and do return from interrupt
        jmp xit
/       if have reached top of circular buffer then reset pointer
/       back to bottom
reset,  tad bstart
        dca ptr2
/       return from interrupt
        jmp xit
/
/
/
/       Page 3,
/               the code of the processing routine
/       its not serious so no comments
*600
proc1,  0
        tad datum
        jms count
        cll rar
        sni
        jms count
```

```
        cia
        tad datum
        cia
        rtr
        sni
        jms count
        cia cll
        jmp i proc1
count,  0
        sna
        jmp i count
        dca word
        dca c1
count1, tad word
        rar
        dca word
        szl
        isz c1
        cia cll
        tad word
        sza cla
        jmp count1
        tad c1
        jmp i count
word,   0
c1,     0
$
```

9. Limitations of the PDP-8 architecture.

The principal advantage of the PDP-8 as an introductory machine is its simplicity. Despite its somewhat clumsy addressing mechanism, the machine is fairly easy to program (perhaps because one can usually remember the entire instruction repertoire(?)). Some disadvantages to the design are obvious. One would like more instructions. It is, for example, tiresome to have to write a subroutine to perform an "or" of two data elements through some contrived sequence of complementing and "anding" of data. Certainly, a few additional instructions would allow for shorter programs.

There are more substantial problems in virtually every aspect of the design. The subroutine call mechanism proves extremely clumsy if one has many arguments and results to pass; it's impossible to have recursion. The response to interrupts is unnecessarily slow; a little extra hardware can make things a lot faster. The fixed size, one word, for every datum is very cramping and leads to clumsy code. The arithmetic facilities are inadequate, there are no multiply or divide instructions in the basic machine, it's difficult to detect arithmetic overflow. The address space is too small, one can't have large programs. The paged addressing scheme leads to inefficient use of even such memory as is available. There are too many restrictions on how one may pass data to and from the arithmetic logic unit.

Many of these problems will be addressed in the lecture course when more sophisticated machine architectures are introduced.

NAME

assemble, exec8, trace8 - prepare and run programs for a simulated minicomputer

SYNOPSIS

/pub/211/assemble name

/pub/211/exec8

/pub/211/trace8

DESCRIPTION

exec8 and *trace8* simulate the execution of programs on a Digital Equipment Corporation PDP-8 minicomputer. *assemble* converts source programs, written in PDP-8 assembly language, into object files for these two simulators.

trace8 is intended to help illustrate the basic "*fetch-decode-execute*" cycle of the machine. It maintains a display of the cpu registers, of the bus connecting the cpu to memory, and of a window into memory.

exec8 is used to illustrate more advanced topics, such as flag- and interrupt-driven i/o. The displays showing the status of the simulated machine are less comprehensive than those maintained by trace8; various components of these displays are optionally selectable. exec8 also incorporates a simple interactive "debugging" package that allows breakpoints to be set and the contents of memory and cpu registers to be displayed.

FILES

/pub/211/source/symbols, symbol table containing standard opcode mnemonics used by assemble.

'.8.kbd.1,.8.tty.1' , these two files (in the user's working directory) are the input for the pseudo-keyboard and output for the pseudo-teletype of the simulated PDP-8. If the program run on the simulator requires input then this must be copied into the file .8.kbd.1 prior to execution.

name, the file containing the user's PDP-8 source code.

object, a file, created by assemble, containing the assembled object code.

dumpfile, a file (optionally created by exec8 or trace8) showing the contents of memory of the machine before and after execution of the simulated program.

SEE ALSO

Course notes describing the simulator displays and the debugging options built into exec8.

BUGS

Please report bugs, by *mail nabg*, as they are found.