Department of Applied Analysis and
Computer Science

Technical Report CS-73-10

April 10, 1973

A PDP-11 SIMULATOR FOR THE H6050

by

David Lamb
University of Waterloo

Faculty of Mathematics

University of Waterloo

Waterloo, Ontario

Canada

Department of Applied Analysis

&

Computer Science

Department of Applied Analysis and
Computer Science

Technical Report CS-73-10

April 10, 1973

A PDP-11 SIMULATOR FOR THE H6050

by

David Lamb
University of Waterloo

A PDP-11 Simulator for the H6050

David Lamb

INTRODUCTION

This document is a description of the PDP-11 simulator on the Honeywell 6050 here at Waterloo. The program simulates the operation of a Digital Equipment Corporation PDP-11 model 20 central processor with twelve kilowords of core memory. Built into the simulator is a supervisor which functions similarly to the ODT-11x debugging supervisor for the PDP-11. The monitor allows the user to examine and modify core locations, to place breakpoints in his machine language program and to run his program.

A description of the internal logic is also provided with special attention to maintenance and extension of the supervisor and to the handling of peripheral simulation. Plans for future capabilities are outlined.

For further information on the PDP-11 and the ODT-11x supervisor, consult the PDP-11 / 15, 20, x20 processor handbook and the PDP-11 paper tape software manual.

The Machine

This section is intended as a general introduction to the PDP-11 for those unfamiliar with it, and may be skipped by those who have had some experience with the machine.

The PDP-11 is a sixteen-bit wordlength mini computer with each eight bit byte individually addressable. It has eight general purpose registers, one of which, register 7, serves as the program counter. One other register, register 6, is used as a hardware stack pointer for the handling of interrupts and supervisor calls. The power of the machine lies in its addressing scheme and in its handling of peripherals and interrupts.

The PDP-11 uses varying-length operation codes to make the best use of the available space in a machine word. There are four basic groupings of operations: double operand instructions, single operand instructions, conditional branches, and a final group which holds all those instructions which do not fit into one of the other three categories.

A PDP-11 operand address consists of six bits. The first two bits indicate the basic addressing mode: the third bit is used to indicate indirect addressing; the last three bits indicate one of the general registers to be used in forming the address. In register addressing mode, (mode bit 00) the indicated register is used as the operand. In autoincrement mode, (mode bits 01) the contents of the indicated register are used to address the operand; after the address is determined, the contents of the register are incremented to point to the next sequential core location. In autodecrement mode (mode bits 10) the contents of the register are decremented to point to the previous sequential core location and are then used as the address of the operand. If the instruction operates upon bytes, updating of register contents is done in steps of one. If the instruction operates on words, or if the register in register 6 or register 7, the updating is in steps of two. In index mode (mode bits 11) the contents of the word following the instruction are added to the contents of the indicated register and used as the address of the operand.

If the indirect bit is a 1, the contents of the address calculated as outlined above are used as the address of a core memory location. The contents of this core word are fetched and used as the address of the operand.

Addressing modes 00 and 11 can be used as in most conventional computers to refer to registers and core location in the ordinary way. Modes 01 and 10 are extremely useful in the manipulation of stacks. On the PDP-11 stacks build downward, higher stack positions being lower in physical core. A MOVE instruction in autodecrement mode can be used to push items onto a stack: in autoincrement mode a MOVE pops items off the stack.

The PDP-11 uses a hardware stack with register six as the stack pointer in the handling of interrupts. The top four kilowords of memory are not core memory, but peripheral device registers. They can be manipulated with all of the instructions used to reference core memory, making specific input-output instructions unnecessary. One of these registers, at location octal 177776, is the processor status word. It indicates the current priority of the central processor, as well as certain conditions such as whether or not the result of the last operation was negative or zero, or resulted in a carry or an overflow. Each peripheral device is assigned an interrupt vector consisting of two consecutive locations in core memory. When the device requests an interrupt, the central processor pushes the current program counter and processor status words onto the hardware stack. It then fetches a new program counter and processor status from the interrupt vector associated with the peripheral device. Supervisor calls are treated in the same manner.

Use of the Supervisor

The supervisor built into the simulator is a powerful tool for debugging machine language programs. It offers the user a wide range of commands for examining, modifying, and running his program.

To begin a session with the supervisor, sign on to TSS on the Honeywell and type RUN CCNG2/PDP11,R at the subsystem selection level. The simulator will identify itself and type an asterisk to indicate that it is ready to accept input. The simulator expects all numerical input to be in octal, and

will type out all numerical responses in octal. The simulator will respond with a question mark to any input it does not understand. It scans characters to any input it does not understand. It scans characters until it finds either an illegal character or a complete command, at which point it executes the command or prints a question mark. Thus, any characters typed after a legal command will be ignored. Any blanks or tabs in the command are totally ignored. To terminate a session with the simulator, hit ATTN when the supervisor asks for input.

The structure of commands accepted by the supervisor is identical to that of ODT-11x commands. There are two basic sorts of commands: special character commands and letter commands. Special character commands are of the form nS where n is null or an octal number and S is a special character. Letter commands are of the form n;mL where n and m are octal numbers or null and L is an uppercase or lowercase letter. Each commands is described individually below.

Commands to Open a New Location

The slash opens the word at location n for examination and modification. The supervisor prints the six-digit address and the six digit contents of location n. If n is omitted the slash opens the last-opened location.

The backslash on ASCII terminals or the or-bar on a 2741 opens a byte at location n. The six digit address and three digit contents are printed. The backslash alone opens the last byte addressed.

For these two commands n may also be the special symbols $B, $M, $P, $S, or $m where m is an octal digit. $Bm opens a core location which holds the address of breakpoint m. (Breakpoints are explained below). $B alone is equivalent to $BØ. $Pm opens a core location which holds the repeat count for breakpoint m $P is equivalent to $PØ. $M opens a location which holds the mask used in search operations. Searches are explained below. $S opens the status word at location 177776 octal. $m opens general register m.

The remaining special character commands interpret the argument n in a different way. If an octal number is specified before these special characters the octal number (modulo 16 bits if a word is open and modulo 8 bits if a byte is open) becomes the new contents of the last-opened core location.

The linefeed character on ASCII terminals or the exclamation mark on 2741's opens the next sequential location.

The up arrow or caret on ASCII terminals or the cent sign on 2741's opens the previous sequential core location.

The carriage return has no effect other than causing the supervisor to ask for a new line of input.

The equals sign interprets the contents of the currently open location as an address and opesn the location so addressed. This is equivalent to the commercial at for ODT-11x. Unfortunately, the at is used as a line-delete on 2741's and as a character delete on ASCII terminals and so a substitution was necessary.

The greater-than sign interprets the open location as a relative branch instruction and opens the addressed location. For PDP-11 conditional branches the low order byte is interpreted as an eight-bit two's complement number, doubled and added to the program counter before being used as a branch address.

The back-arrow or underscore takes the contents of the open location, adds the contents of the program counter, and opens the addressed location.

The less-than sign opens the location open prior to the last equal sign, back arrow, or greater-than sign commands. The less than sign can recover from up to eight such changes of sequence, after which it behaves like the line-feed key until the next equal sign, back arrow, or greater-than sign is received.

The letter commands are described below. The B command is used to manipulate breakpoints. A breakpoint is a location in a user's program where the user's program is interrupted and control goes to the supervisor. The supervisor allows up to eight breakpoints, each identified by an octal digit.

The command n;mB sets breakpoint m at location n. If m is omitted, the supervisor searches for a free breakpoint. If it cannot find one, it prints the message NO FREE BPT and waits for a new command. If a free breakpoint is found it prints out the number of the breakpoint that was found. The command ;mB removes breakpoint m. The command ;B removes all breakpoints.

The command n;E searches for all words which address location n. All words which contain the value n, which are relative branches to n, or which when added to the program counter at that point refer to n are printed. The search is done between the limits specified by the contents of $M+2 and $M+4. These lcoations can be accessed by typing $M and a slash, then using the linefeed key.

The command n;G runs the user's program starting at location n. The program will continue executing until a breakpoint is reached, the ATTN button is pressed, or an error of some sort occurs. When a breakpoint is reached the supervisor prints the breakpoint number and the address at the breakpoint.

The command ;P is used to continue executing a program once a breakpoint has been reached. The command n;P continues until the breakpoint has been reached n times. These repeat counts are kept in the location $P though $P+16 octal.

The command n;W searches for all bit patterns matching n in the positions specified by $M between the addresses specified by $M+2 and $M+4. Comparison is done by taking the exclusive or of n and the compared location and taking the end of this result with the contents of $M. Thus to find all locations whose second octal digit from the right is a 2, one would set $M to 70 and give the command 20;W.

The command n;mO calculates the offset of location n with respect to location m. If m is omitted, the offset with respect to the current location is taken. If the offset is even and in the range -256 to +255 decimal, the eight bit offset is printed in addition to the sixteen bit offset. This is used to calculate the offset used in relative branch instructions. For example if we are examining location 346 and give the command 414;0 the sixteen bit offset is 44 and the eight bit offset is 22.

The command n;mM enables memory protect on mode n through m. During execution of the user's program, whenever reference is made to a protected location the simulator prints a message to that effect, prints the current instruction the contents of all the general registers, and the protected address, and waits for a command from the typewriter. n;M protects the single location n. The command ;M enables the memory protect option if it has been disabled.

The command n;mC clears memory protect on location n through m. If m is omitted the location n alone is cleared. If both n and m are omitted, memory protect is disabled. A subsequent ;M will re-enable memory protect, protecting all locations protected before the ;C command unless some location were cleared by an intervening n;mC.

Memory protect is normally enabled. The locations $0 through $7, $B0 through $B7, $PØ through $P7, $M, $M+2 and $M+4 are all normally protected.

Internal Logic of the Simulator

The simulator was originally written entirely in Honeywell Fortran. Later, routines that were inefficient in Fortran were rewritten in GMAP, the assembly language of the Honeywell 6050.

The main program is written in Fortran. It consists logically of two sections, the command handler and the hardware simulator. Statement numbers in the 1000 range comprise the command handler; statement numbers in the 2000 range comprise the simulator. In addition there are several statements in the 9000 range which print out certain error messages.

The core memory of the simulator is a large integer array, MEM. The Honeywell word is thirty-six bits in length whereas the PDP-11 word is only sixteen bits in length. To avoid problems of wasting core storage, two PDP-11 words are packed in each Honeywell word, one to each eighteen bit half. The PDP-11 word is right-justified in the half word. The high order bit is set to a one if this location is memory protected and a zero otherwise. The second bit is a one if this location is a breakpoint and a zero otherwise.

For quick access, the general registers each use full Honeywell word.

Eventually, the entire program will be written in GMAP. When this is done, index registers will be used for the more frequently used PDP-11 registers, registers six and seven and possibly register five. (PDP-11 standard conventions use register five for subroutine linkage).

Peripheral registers are implemented differently. When the simulator recognizes that a peripheral register is being referenced, it sets switches so that reads and writes are not done by the standard routines. A special routine is called for each peripheral register that is implemented. At present, only the Processor Status register at location 177776 is implemented. This approach is taken for two reasons. First, special routines are needed for reading and writing peripheral registers because such reads and writes have side effects. For instance, loading the teleprinter data buffer causes the loaded character to be printed on the teleprinter; setting the low order bit of the disk control register indicates a disk operation. The second reason for this approach is that when few peripherals are implemented it saves approximately two kilowords of Honeywell core storage. At present the simulator takes approximately eight kilowords of core. At peak points during the day this makes response time very slow, since TSS swaps out large programs as frequently as it can.

The logic of the main program is quite simple. The supervisor calls a subroutine to fetch and decode a command from the typewriter. It then branches to the appropriate command routine to execute the command. Most commands return for the fetch of another command after they have finished executing. The ;G and ;P commands branch to the hardware simulation section after loading registers 0 to 6 and the "breakpoint proceed" counts with whatever new values were specified by the user. Any changes the user makes to the breakpoint address section are ignored; the addresses are kept in an array inaccessible to the user and hence cna be changed only by the ;B command. When the hardware simulator returns to the supervisor, the special core locations have the contents of the general registers, the breakpoint addresses, and the proceed counts written into them.

The hardware simulator fetches the word of core storage pointed to by register 7 and interprets it as an instruction. It branches to the appropriate routine to handle each instruction or to an error message routine if the fetched word was not a legal instruction. At the completion of each instruction a section is executed which sets the appropriate condition code bits in the processor status word depending on the results of the instruction.

The subroutine LOCATE translates the address it is given into a subscript for MEM. The word MEM in which the PDP-11 word is located is placed in location MAR. The lower bit of the address, which indicates whether to take the add or even byte, is placed in switch ODD. The second bit, which indicates in what half of the Honeywell word the PDP-11 word is located, is placed in the switch TOP. The routine checks to see if the address is longer than 12K. If so, it checks to see if it is in the peripheral range. If it is in the peripheral range, the routine does a table lookup to see if a peripheral is assigned to that location. If the address is greater than 12K and is not a legal peripheral register address, an error return is taken. If it is a legal peripheral address the address of the routine to handle the peripheral is placed in location PERRTN.

The table used to look up peripheral register addresses starts at location PERTAB. Table entries are of the form

```
VFD         18/xxxx, 18/ADR
```

where xxxx is the affect of the peripheral register address in the peripherals bank, divided by two and ADR is the address of the routine associated with the peripheral register. The routines are of the form

```
TRA (read register section)
(instructions for write register section)
```

At present the interrupt structure is not implemented; when this is accomplished, there will be a third word just previous to these two

```
TRA         (INIT pulse received)
```

The INIT pulse is generated by the RESET instruction and restores all devices on the bus to their status at power-on.

The routine FETCH picks up the PDP-11 word indicated by the MAR and TOP registers. It checks the memory protect and breakpoint bits of the addressed word, placing these in the switches MPT and BPT respectively. If the PERIP switch is set it branches to the read routine for the peripheral register being addressed.

The routine FILL writes a new value into the addressed word, saving the current MPT and BPT bits. Both FETCH and FILL ignore the ODD switch. That is, they both operate on full PDP-11 words.

The routines GET and PUT check the switch BYT to determine whether a byte or a word is to be operated upon. Thereafter they operate like FETCH

and FILL, save that in byte mode they operate only upon a single byte. In word mode, if the ODD switch is on, GET takes an error return since all word data must be taken from an even address. PUT does no such checking because a GET is always done before a PUT, to check for errors.

The routine BUMP updates TOP and MAR to address the next sequential PDP-11 address. It is called in situations where it is known addresses will be sequential, such as in word searches with the;W command. BUMP does no checking for overflow into nonexistent memory, so this must be done by the routine which call it.

The routine ORDERS reads in and decodes commands from the typewriter. It uses the ASCII code for each character to look up in a table the internal code for each character. The top two bits are used to indicate what type of letter has been found. Type bits 00 indicate a number; bits 01 indicate a letter; bits 10 indicate a special character; and 11 indicate a character which is not passed to the calling program but is dealt with inside the command decoder. All illegal characters are represented as special character octal 412. Special characters presently implemented are represented (minus type bits) as octal 01 through 11. The letters are represented as octal 01 through 11. When a letter is recognized, octal 12 is added to it to give a command number. If a letter was not expected, the register is cleared to 00 before 12 is added, thus giving the command number 12 which indicates an illegal character. If more special characters are to be implemented, all locations in the translation table TABLE which contain octal 412 must be updated to contain 4xx where xx is 1 + the number of special character implemented. Location LETTER +1 must be updated to contain the instruction ADX2 = 0XX. Adding new letter commands is easier: the locations in TABLE corresponding to the letter in both upper and lower case should be changed from 412 to 2nn where nn is one greater than the number of the last letter command implemented. Currently, nn would be octal 12. An entry would have to be added to the end of TAB2 to indicate whether this letter can follow the $ to indicate a special location. Entries in TAB2 are zero if the letter cannot follow the $ and contain the address to be examined otherwise. The top bit is set if the letter can be followed by a single digit as with $Pn and $Bn, and is zero otherwise.

The routine MSG prints out messages and reports. If the first argument is negative, MSG simply uses the argument to pick a tally out of a table to reference the message to be printed and print the indicated message. For zero or positive arguments, MSG branches to a routine which packs additional information into a line before printing it out. Thus, single messages which remain the same constantly are referenced by negative arguments; messages which change, such as the message which prints out the contents of a core location, are referenced by positive arguments.

The routines SETFLG, SETBRK, and UNSET are used to deal with the ATTN key. SETFLG sets up a swtich which is set TRUE when the attention key is pressed. SETBRK causes the program to resume execution where it was interrupted when attention is pressed. This is done so that an "instruction" (that is the routine which executes a PDP-11 instruction) is not interrupted in the middle. At the beginning of each fetch cycle the break flag is checked. UNSET causes the hitting of the attention key to terminate the simulator and return control to the TSS subsystem from which the simulator was called.

The routine ADRESS decodes the six-bit PDP-11 operand address. If the processor is in word mode or if the specified register is register six or register seven, the increment to be used is autoincrement or autodecrement addressing is set to two; otherwise, it is set to one. The routine decides what addressing mode to use based upon the first two of the six bits. Register mode without indirect addressing sets switch REGM to TRUE, otherwise it is set to FALSE. If indirect addressing is specified, a FETCH of the addressed core location is performed (in register mode, the contents of the register are used). If autodecrement addressing was specified for register six, and if the result left a value less than octal 400 in register six, the stack overflow switch STAK is set. This switch is tested at the beginning of each instruction fetch cycle.

The routine TRAP performs the standard interrupt sequence. It stacks the current processor status and program counter and picks up the new processor status and program counter from the core address specified by its argument.

This completes the description of the major routines of the simulator. There are a number of minor routines which are either mostly self-explanatory or non-essential to an understanding of the workings of the simulator.

Limitations of the Simulator

The trace trap and T-bit features (for which see the processor handbook) are not implemented. However, these features are used only by ODT on the PDP-11 and so should not be needed due to the capabilities of the supervisor.

The interrupt structure and peripherals are not yet implemented. These features had to wait until the FETCH and FILL routines were written in GMAP, as Fortran would have been extremely inefficient in the handling of such features. The routines to handle these features are already planned out: however, there remains no time in the term to implement and debug them.

The simulator presently gives extremely slow response when TSS is crowded, since it requires a large area for the core array. TSS swaps such large programs as frequently as possible to make room for smaller routines.

Error conditions such as nonexistent memory, stack overflow, and addressing errors currently cause error messages to be printed, whereupon control goes to the supervisor. Eventually there will be an option for each error of whether to print the message or to perform the hardware trap sequence defined for each error. At present there is no way for a PDP-11 program to detect and process such errors.

There is at present no way to read in a PDP-11 machine language program save by typing it in at a terminal. Eventually the simulator will be part of a general PDP-11 subsystem which will allow a user to assemble programs, link them with already existent load modules both on the Honeywell and on a PDP-11 connected by a communications line, and run them using the simulator. When the full subsystem is developed, the simulator should provide a powerful tool for software development for the PDP-11 as well as an excellent aid for a course on minicomputers in general and the PDP-11 in particular.

Sample Terminal Session

Accompanying this document is a complete session with the simulator, from signon to signoff. This section describes the session. Using the simulator requires a knowledge of the machine language of the PDP-11, or at least a reference manual close at hand, since all instructions must be typed in an octal rather than in symbolic form. This was not done merely to avoid symbol decoding and encoding, but also to emulate the ODT-11x supervisor which has the same requirement.

The first activity with the simulator is the testing out of a machine language program to multiply two numbers. The two original sixteen bit numbers are placed in registers R3 and R4. The thirty two bit result is returned in R0 and R7.

In the notation of the PDP-11 assembler, PAL-11A, the program is as follows:

```
        CLR RØ
        CLR R7      Zero out registers zero to two
        CLR R2
        MOV #20,R5  Place a count in register 5 of octal 20
LOOP    ROR R4      Rotate R4 right. The lower bit enters the carry indicator
        BCC NEXT    Branch if Carry Clear to NEXT, skipping
        ADD R3,R1   the add instructions.  Add R3 to R7
        ADC RØ      Add the carry bit to register zero.
        ADD R2, R0
NEXT    TST R4      Get the indicators based on the contents of R4
        BEQ OVER    If zero we are finished
        ASL R3      Shift R2 R3 left one place.  The high order bit
                    enters the carry indicator, where the Rotate Left
        ROL R2      moves it into the lower bit of register two.
        DEC R5      Decrement the counter and branch back to the
        BNE LOOP    loop if it is nonzero
OVER    HLT         Halt when done.
```

At LØ1 we begin typing in the machine language equivalent of the above program standing at location 1000. The exclamation point at the end of each line tells the simulator to open the next sequential location. At LØ2 we realize we have made a mistake in the previous line, as we go back up one location to fix it. At LØ3 we must place a branch back to the beginning of the loop at location 1012, so we ask for the offset of 1017 from the current location with the ;0 command. We then type in the branch instruction and then check to see if the branch back at location 1026 was correct. We say 1Ø26; 1Ø36 thinking that this will give us the offset of location 1036 from location 1026. When

we get a negative offset we realize that the arguments should be in the other order. We are puzzled that the offset is three instead of four as we expected until we notice that we should have asked for the offset of location 1040 with respect to 1026.

At LØ4 we set breakpoint 0 at the halt instruction at location 1040. We then apply a standard trick used to print sequential core locations. We open the word containing the mask, $M, and leave the mask at Ø. We change the low limit of the search to the first location to be printed, location 1000, and we set the high limit of the search to the last location to be printed, location 1036. When we do a word search, all bits will be masked off and so the search will succeed for every word in the range and consequently every word will be printed out. At LØ5 we give the search command, which proceeds to print out our program.

At LØ6 we examine register zero, with the intent of proceeding to examine the other general registers. We realize two lines later that it is smarter to use the above-mentioned word search technique. At LØ7 we place two numbers to be multiplied in registers three and four (each contains a three). At L08 we give the command to begin executing our program. The line just after our command is the simulator telling us we have encountered breakpoint zero, at location 1040. At LØ9 we give the command to the word search to print out all eight general registers. To our horror we discover that the register zero register one combination does not contain the octal equivalent of nine, as we would have expected, since 3 times 3 gives 9. Each register contains nine, whereas we expected register zero to contain zero. We examine the printout at the program to discover we have mistyped an instruction.
Instead of 103002 (Branch on Carry Clear to NEXT)
we typed 10302 (Move register three to register two)
At L10 we correct our mistake, reinitialize registers three and four, and run the program once more. The supervisor tells us we have reached breakpoint zero at the end of the program and we ask for the contents of the general registers. Lo and behold, we have the right answer! At L11 we try again for two new numbers, octal 17, decimal 15. We try again and find the answer to be octal 341, which is decimal 225 as we would hope.

At L12 we begin to test a few other capabilities of the simulator. We examine the status word and register seven, the program counter. The program

status word has the zero bit set. At L13 we see that the program counter is 1040 and we ask to examine location 1040 with the equal sign command. At L14 we remove breakpoint zero and try to begin execution at location 1040. This location has a HLT instruction.

At L15 we place memory protect on location 2000 to 2012, and try to execute location 2000; the memory protect mechanism traps us. The same occurs when we try to execute location 2012; however, location 2014 is not protected so it is executed.

At L16 we open the current location which is the address pointed to by the program counter. We then give the command to open the location open prior to the last equal sign command; that is, the location holding the program counter, register seven. We then change the high and low search limits to bracket our program again, and search for all references to location 1040 at L17. We then search for all references to location 1012.

At L18 we set a breakpoint at location 1012. The simulator tells us it has assigned breakpoint number zero. We set registers three and four to five and six respectively and begin executing our program. We hit the breakpoint, which we placed on the instruction at label LOOP in the program. We set the search limits around the general registers again and examine the registers. At L19 we give the command to proceed from this breakpoint. We reach the breakpoint again and ask for the register again. At L20 we tell the simulator to proceed, ignoring this breakpoint twice. The program completes and reaches the halt instruction at the end. We print out the registers again, note that our result is octal 36 or decimal 30 as we would expect. We then hit the ATTN button to exit the simulator and logout from TSS.

```
        125221
        UNTELOG 6050 04/09/73 10:03  7060
        USER ID --user1
        0 BLOCKS FILE SPACE AVAILABLE


        SYSTEM ?runv ccng2/pdp11,r


        pdp-11 simulator

L01:    *1000/
        001000/000000  *5000!
        001002/000000  *5001!
        001004/000000  *5002!
        001006/000000  *11705!
        001010/000000  *¢
L02:    001000/011705  *12705!
        001010/000000  *20!
        001012/000000  *6004!
        001014/000000  *10302!
        001016/000000  *60301!
        001020/000000  *5500!
        001022/000000  *60200!
        001024/000000  *5704!
        001026/000000  *1404!
        001030/000000  *6303!
        001032/000000  *6102!
        001034/000000  *5305!
L03:    001036/000000  *1012;o
        177752    305
        *⁄
        001036/000000  *1305!
        001040/000000  *
        *1026;1036o
        177756    373
        *1036;1026o
        000006    003
L04:    *1040;0b
        *0m/
        057772/000000  *!
        057774/000000  *1000!
        057776/000000  *1036
```

```
L05:    *;W
        001000/005000
        001002/005001
        001004/005002
        001006/012705
        001010/000020
        001012/005004
        001014/010302
        001016/060301
        001020/005500
        001022/002200
        001024/005704
        001026/001404
        001030/000303
        001032/000102
        001034/005305
        001036/001305
L06:    *$0/
        057752/000000   *
        *$
         05L
        $m/
        057772/000000   *!
        057774/001000   *57752!
        057776/001030   *57770
        *;W
        057752/000000
        057754/000000
        057756/000000
        057760/000000
        057762/000000
        057764/000000
        057766/000000
        057770/000000
L07:    *$3/
        057772/000000   *3!
        057774/000000   *3
L08:    *1000;G
        L0: 001040

L09:    *;W
        057752/000011
        057754/000011
        057756/000000
        057760/000000
        057762/000000
        057764/000017
        057766/000000
        057770/001040
L10:    *1014/
        001014/010302   *103002
        *$7/
        057766/000005   *3!
        057772/000000   *3
        *1000;G
        L0: 001040
```

```
        *;~
        057752/000000
        057754/000011
        057756/000000
        057760/000000
        057762/000000
        057764/000017
        057766/000000
        057770/001042
L11:    *^%/
        057770/000000   *17!
        057772/000000   *17
        *1000;~
        L0: 001042

        *;~
        057752/000000
        057754/000341
        057756/000000
        057760/000170
        057762/000000
        057764/000015
        057766/000000
        057770/001042
L12:    *$<
        177776/000004   *$7/
L13:    057770/001040   *=
L14:    001040/000000   *;0b
        *1040;~
        halt    000000 001042

L15:    *2000;2112m
        *2000;~
        memory protect
        instruction 000000
        registers 000000 000341 000000 000170 000000 000000 000000 000000
        address 002000

        *2112;~
        memory protect
        instruction 000000
        registers 000000 000341 000000 000170 000000 000000 000000 000000
        address 002112

        *2014;~
        halt    000000 002016
```

```
L16:    */
        [illegible]   *<
        057770/002016   *$m/
        057772/000000   *!
        057774/057752   *1000!
        057776/057770   *1036
L17:    *1040;e
        001026/001504
        *1012;e
        001036/001305
L18:    *1012;b
        0
        *$3/
        057700/000170   *5!
        057702/000000   *6
        *1000;c
        L0: 001010

        *$m/
        057772/000000   *!
        057774/001000   *57752!
        057776/001036   *57770
        *;'
        057752/000000
        057754/000000
        057756/000000
        057760/000005
        057762/000006
        057764/000022
        057766/000000
        057770/001012
L19:    *;p
        L0: 001010

        *;'
        057752/000000
        057754/000000
        057756/000000
        057760/000012
        057762/000003
        057764/000017
        057766/000000
        057770/001012
L20:    *2;p
        halt    000000 001042
```

```
*;^
057752/000000
057754/000036
057756/000000
057760/000024
057762/000000
057764/000010
057766/000000
057770/001042
*
SYSTEM ?logout
**resources used $   2.00, used to date $   371.50= 44%
**time sharing off at  10.400 on  04/00/73
```