A COMPARISON OF TWO MICROPROGRAMMABLE PROCESSORS:

PDP-11/40E AND MLP-900.

John D. Oakley
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa. 15213

May, 1975

ABSTRACT

A study was undertaken to evaluate the capabilities of two microprogrammable processors: the MLP-900, a vertically-encoded 36-bit machine at the Information Sciences Institute and available over the ARPA Network; and the PDP-11/40E, a horizontally-encoded 16-bit microprocessor at Carnegie-Mellon University. The paper presents a description of the two machines, and compares their performance on a number of benchmark programs (including an emulator for the NOVA computer). In addition, the machines are compared along dimensions of two-way conditional branch costs, basic architecture, and difficulty of programming. The PDP-11/40E performed between 10% and 25% faster on all the benchmarks except the multi-word integer multiply, where the MLP-900 was four times faster (because of its wider data path).

This work is supported by the Advanced Research Projects Agency (ARPA) of the Department of Defense, under contract F44620-73-C-0074, monitored by the Air Force Office of Scientific Research.

A Comparison of Two Microprogrammable Processors

I. Introduction.

The ongoing project at Carnegie-Mellon University investigating the symbolic manipulation of computer descriptions (SMCD) has the following primary goals: to design a language able to describe precisely an arbitrary target machine in a convenient fashion [1]; to create a compiler for this language which will transform a source machine description into a more usable representation; and to utilize this compiled machine description in a range of applications, from computer-aided machine design to efficient compiler-compilers [2]. A multi-level simulator for the target machine, capable of operating on the gate level, regiser-transfer level, or functional level, is also being designed, and will be important in many of the applications. This simulator will be implemented on a microprogrammable processor for efficiency reasons. Eventually, an optimizing micro-compiler will be developed whch will automatically produce an efficient simulator directly from a compiled target machine description.

Two possible machines for the simulator are currently being considered. The first is the MLP-900, a vertically-encoded microprogrammable processor which exists at the Information Sciences Institute at USC and is available over the ARPA Network. The second is a PDP-11/40, a horizontally-encoded microprogrammable processor, which has been modified at CMU to include a read/write control memory and other hardware extensions to make it a general purpose microprogrammable machine. The extended machine is known as the PDP-11/40E.

The MLP-900 was originally designed by the Standard Computer Corporation, and has been described in [3]. The machine never reached production, though a single prototype was constructed which now exists at ISI. Various hardware modifications

have been incorporated in the MLP-900 at ISI, and the machine is described in [4]. The MLP-900 is linked to one of ISI's PDP-10s (which runs TENEX and is on the ARPA Network). Software support exists to enable easy access to the MLP-900 from the PDP-10, including a high-level assembler, debugging facilities, and I/O routines. To facilitate multiple users of the MLP-900, a protected "microvisor" program resides in the MLP's microstore to handle swapping of user microprograms. This mini-supervisor also handles I/O requests from the MLP to the PDP-10. Since the MLP uses the main memory of the PDP-10, the TENEX memory management features of virtual memory and paging are incorporated in the microvisor. Page table maintenance and PDP-10 communication are transparent to the MLP user.

The PDP-11/40E constitutes an ongoing research project in itself [5], and documentation and support are not necessarily complete. At the time of writing, there is a micro-assembler and a simulator available on the PDP-10, with a link available to transmit the assembler output to the PDP-11/40E. Also, a prototype micro-debugging package exists to aid hands-on debugging. Work is currently in progress to incorporate the PDP-11/40E into C.mmp, CMU's multi-miniprocessor system [6].

Table 1 enumerates the major parameters of the PDP-11/40E and MLP-900 microprocessors.

II. Machine Description of the MLP-900.

The MLP-900 is a synchronous, vertically-encoded microprogrammable machine, with a fixed clock cycle of 250 nsec. In appearance, its instruction set is similar in structure to the instruction set of a regular general-purpose machine; hence, programming it is, at least superficially, reasonably straightforward.

The MLP is separated into two distinct parts. Each part has its own processor, instruction set, and data storage (registers). The two parts are designated the "Operating Engine" (OE) and the "Control Engine" (CE). The basic idea behind this approach is to factor out independent functions (ie, arithmetic and control) into distinct pieces of hardware -- and consequently achieve a low-level parallelism by being able to execute two microinstructions simultaneously.

The MLP-900 has 4K of 32-bit memory for microinstructions in which both OE and CE instructions reside. A conventional program-counter is used to address the next microinstruction to be executed. The op-code field of each word contains a bit which indicates to the fetching unit whether the instruction should be routed to the OE or the CE for decoding and execution. However, whenever the program-counter points to an OE instruction followed immediately by a CE instruction, the effect is to fetch both and route them to their respective engines simultaneously. Parallel execution of the two instructions is thus achieved (except in certain cases of interaction where an extra cycle is required). There is no restriction on how the OE and CE instructions are arranged in the micro-instruction memory, but if the entire microprogram is arranged in OE-CE pairs, the effective speed of the program will be doubled over a (hypothetical) single-engine machine. See Fig. 1 for a schematic diagram of the MLP components.

The Operating Engine contains the following memory components:

- 32 x 36-bit general purpose registers (with octal designation R.0 - R.37).
- 32 x 36-bit mask registers (M.0 - M.37).
- 1K x 36-bit 200 nsec. auxilliary memory (A.0 - A.1777).
- 32 x 36-bit miscellaneous registers with dedicated functions (MISC.0 - MISC.37).

The OE also contains hardware to perform the following functions:

Arithmetic/logical operations on a 36 bit data path, including shifting-masking operations (GEAR, SHIN instructions).
Communication with main memory (CEDE instructions).
Intra-OE data transfers (GENT instructions).

All of the OE memory is available to the microprogram user, with the exception of most of the MISC registers and 16 of the mask registers (which are only accessible in microvisor mode).

The Control Engine contains the following memory components:

256 individually addressable flip/flops (F.0 - F.377).
16 x 8-bit pointer/counter registers (P.0 - P.17).
16 x 16-bit subroutine-stack registers (S.0 - S.17).
8 x 16-bit miscellaneous registers with dedicated functions (CE.60 - CE.77).

The CE also contains hardware to perform the following functions:

Conditional and unconditional branches (BRAT, BEAD instructions).
Conditional and unconditional subroutine calls and returns (BENT, BORE, BEAD instructions).
Flag manipulation (MAST instructions).
Loop control operations (BRAD, BLOT instructions).
Intra-CE data transfers (MOVE instructions).

For consistency in referencing, all the CE memory is addressable as 96 8-bit registers (named CE.0 - CE.137), or 16-bit double-registers. A considerable portion of the CE memory has dedicated functions, representing various internal states of the MLP and is, frequently, writable only in microvisor mode. However, the rest of the CE memory is for general use by the microprogram user.

For the most part, the OE and CE have little interaction (which was clearly a design decision). There are a couple of exceptions, however: OE ALU instructions (GEAR and SHIN) have side effects on status flip/flops in the CE, and can use a pointer register (P.0 - P.17) in their operation. Also, an Exchange Bus (XBUS) is addressable by both engines, and is used to explicitly transfer data between the two engines. One cycle and an OE/CE instruction pair (GENT/MOVE) are necessary to do this.

Two CEDE instructions are required to perform a memory reference on the MLP-900: a fetch-operand/wait-for-operand (FOP/WOP) pair for input; and a set-address/store-operand (SAD/SOP) pair for output. Intervening instructions can occur between each pair to effect overlap with memory access time. Memory access time is approximately 700 nsec. with the current configuration on the MLP-900, including page table translation time. In addition, 600 nsec. total is required for the execution of the CEDE pair, for a total of 1300 nsec/memory reference. Three 250 nsec. cycles can usefully be overlapped during the reference.

Appendix A gives some examples of MLP-900 microinstructions.

III. Machine Description of the PDP-11/40E.

The PDP-11/40 is a horizontally-encoded microprogrammable machine with basic cycle times of 140 to 300 nsec. and a data path width of 16 bits. The unmodified machine has a 256-word x 56-bit ROM which contains microcode to implement the standard PDP-11 instruction set.

Though not designed explicitly for modification for general purpose operation, a model 40 at CMU has been modified to include the following:

1K x 80-bit read/write microstore (instructions and data).
16-word x 16-bit hardware stack.
General field extraction (i.e., shift/mask).
Expanded carry propagation logic.

The enlarged microword (80 bits instead of 56) is for control of the additional hardware. See Fig. 2 for the internal data paths of the PDP-11/40E.

Each microinstruction contains the following information:

(a) For each multiplexor (i.e., DMUX, BMUX, BAMUX, SMUX, etc.) a field which selects one of the possible sources for that multiplexor. (Thus, these fields control what is on the various buses.)

(b) For each register (i.e., B, STK, D, any one general register, etc.) a field which indicates whether that register is selected (for loading) from the specific source bus it is connected to.
(c) A next-address field giving the address of the next microinstruction to be executed (i.e., an embedded 'GOTO' field).
(d) Miscellaneous fields controlling such things as:
ALU function (add, subtract, AND, OR, etc.)
Stack function (push, pop, reset pointer, etc.)
Shift/mask specification
RAM read/write
Unibus read/write
Carry propagation
Conditional branching
Clock pulse generation

There are three basic clock pulses possible during each microinstruction:

P1, at t=140 nsec., enables possible loading of B, PS, R[i] (i.e., a single general register) from the DMUX bus, and BA from the RD bus.
P2, at t=200 nsec., enables possible loading of D and BA registers from the ALU output.
P3, at t=300 nsec., enables possible loading of the B, PS, R[i] registers from the DMUX bus. This pulse is similar to P1, without the BA enabling.

Each microinstruction contains a field designating the pulse sequence to be used in its execution. There are only three possible sequences:

CLK=1 generates a P1 pulse (at t=140 nsec).
CLK=2 generates a P2 pulse (at t=200 nsec).
CLK=3 generates a P2-P3 pulse (at t=200 and t=300 nsec).

The stack and Unibus are "loaded" (i.e., written if selected) at the end of the microinstruction execution, regardless of pulse sequence. Note that it is impossible to specify a P1 pulse and a P2 pulse in the same instruction.

The micro-assembler for the 40E normally computes what pulse sequence is required for each instruction. However, knowledge of the clock mechanism is necessary to understand what functions can be combined in a microinstruction. For example, it is not possible, in a single instruction, to load a general register R[i] from D,

and then perform an ALU operation into D. (The non-existent P1-P2 sequence would be necessary for this. While this sequence might be useful to have, it was not included in the original design of the PDP-11/40 microprocessor by Digital Equipment Corporation. Presumably cost effective or technical reasons precluded its inclusion).

The control memory RAM can be read and written as data, via an address on the top of the stack. However, this can not be done simultaneously with fetching a new microinstruction from the RAM, so some special juggling of where the next microinstruction is coming from (while the RAM is being read/written) must be done. This tends to limit the usefulness of the RAM for storing data.

A memory reference on the PDP-11/40E requires two microinstructions, though other functions can be performed in the same instructions. Since the Unibus of a PDP-11 is asynchronous, explicit synchronizing must take place in the processor during a memory reference. The method used is to supply a "clock-off" bit in the microinstruction which stops the processing of subsequent microinstructions when set. When the Unibus completes its activity, it supplies a signal which restarts the processor clock. Overlap of microinsruction processing with memory references is done simply by waiting to turn the clock off until two or three instructions after the start of the memory reference. However, since the Unibus restart signal is not latched, the clock-off operation must be done before the restart signal is received, or else the processor will hang up.

Memory read access time on the PDP-11/40 machines is, on the average, 700 nsec.; this value can fluctuate from about 500 to 900 nsec., however, depending on the time since the last memory reference. (The longer the interval since the previous access, the less time required for the current reference.) Since two instructions are

required for a memory reference (both reads and writes), the minumum overall time for a reference is the access time plus two P1-cycle instructions, or 780-1180 nsec.

A conditional branch is performed on the PDP-11/40E in the following manner. Assume the current microinstruction is at address A in the microstore, A's successor is at address B, and B's successor is at address C. The desired condition is tested in the current microinstruction at A, with a result of 1 or 0, if the condition is true or false. This bit is temporarily ORed into the next-address field of the following microinstruction, located at B (the condition result is generated too late in the cycle to do the OR with A's next-address field). Assuming that C is an even address, the effect then is to branch to either address C or C+1, if the condition is false or true, respectively. Note that the microinstruction at B is required, even if it is only a NOOP.

The "condition bit" can actually be selected from a data word on the top of the stack via the shift/mask unit. In fact, n bits can be selected, and temporarily ORed into the successor's next-address field to effect a 2^n-way branch. In this case, the lower n bits of address C in the above example must be 0 for the case branch to perform correctly. Such bookkeeping is conveniently left for the micro-assembler.

See Appendix B for some examples of PDP-11/40E microinstructions.

IV. Performance Comparison.

Four primary benchmark programs were written for each machine to compare their performance on specific tasks:

(1) MMPY: 64-bit unsigned integer multiply (128-bit result).
(2) TRANS: Translate routine for a string of 8-bit characters.
(3) ALLOC: Memory allocation routine using Knuth's boundary-tag method [7].
(4) NOVA: Emulator for the NOVA minicomputer (without interrupts or I/O).

The four benchmarks were chosen for various reasons: MMPY compared multi-word arithmetic on the two machines; TRANS exercised their byte-handling capabilities; ALLOC compared their performance on a list-processing application; and NOVA compared their performance on a real-life emulation task.

Each benchmark went through a writing, debugging, improving sequence several times. Substantial improvements were made in the programs over the course of several weeks, as various tricks or better ways of performing the same function were recognized. (This was especially true for the PDP-11/40E microinstructions.) Hence, there is some hope that the benchmarks represent an example of fairly good microcode for both machines.

For the MMPY, TRANS, and ALLOC benchmarks, an attempt was made to keep the number of microinstructions to a minimum on both machines, consistent with at most only a small time increase. This is obviously of great importance on the PDP-11/40E with its smaller amount of microstore. For the NOVA benchmark, however, the emphasis was solely on speed, except where space could be saved at no cost in speed. The NOVA benchmark constituted a complete task so that the whole microstore was assumed available. The first three benchmarks, however, might well be considered small parts of a much larger program, hence attempts were made to conserve a possibly scarce resource.

The MLP-900 programs were assembled, and run on the actual machine at ISI via the ARPA Network. The MLP does not have a timer available to the program, so timing data was computed by hand, including estimates for the memory reference time.

All four of the PDP-11/40E programs were assembled and run on the simulator available on a PDP-10 at CMU, and two (TRANS and NOVA) were also run on the actual

machine. The simulator was more convenient for debugging and timing than the actual PDP-11/40E, so this method was preferred. The execution time included fairly good estimates on the memory reference time.

Tables 2 to 5 show the results of the benchmarks on the two machines. The MMPY program, shown in Table 2, leaves the PDP-11/40E at a severe disadvantage because of the PDP-11's small data path width. The program was a double-word multiply on the MLP-900 and a quadruple-word operation on the 40E. There was sufficient fast registers in both microprocessors to hold all the intermediate results, though this resource was being stretched on the PDP-11/40E. The MLP-900 performed far better on the multiple-word task: slightly more than one third as many instructions and over four times faster than the PDP-11/40E.

On the TRANS benchmark, shown in Table 3, the PDP-11/40E was about 25% faster than the MLP-900. This was not unexpected, since the PDP-11 machine is oriented towards 8-bit bytes, and hence the PDP-11/40E microprocessor has some features which facilitate byte-handling: byte swapping and byte-writes with registers, a write-byte operation to memory, etc. The MLP-900 has no special byte-handling operations outside of its general mask/shift capabilities. These capabilities have certain restrictions, however: shifts of only a few specific values are allowed (thus possibly requiring more than one instruction for a single shift); also, while a field can possibly be isolated from a register in one instruction, storing a value into an arbitrary field always takes at least two microinstructions. Indeed, a four character/word version of TRANS for the MLP-900 was longer and ran slower than its two character/word counterpart in Table 3, even though fewer memory references were required. The extra time came from the slightly awkward field extraction/storing methods required.

Table 4 shows that the PDP-11/40E was slightly faster than the MLP-900 on the ALLOC benchmark. No particular architectural shortcomings of either machine were exercised by this benchmark since mostly full-word operations were used, with 16 bits being an adequate width. However, a fair number of distinct memory reference sequences were necessary in the program. These turn out to be generally less costly on the PDP-11/40E, both in instructions and in time, than on the MLP-900. Also, it is worth pointing out that the MLP-900 did not perform as well when the information was packed into fewer words. For example, an attempt to pack the forward and backward list links into the two halves of a single word increased the number of instructions and memory references over having them in separate words (the version in Table 4). This suggests that the wider data path of the MLP-900 may be most useful when the data in a word is homogeneous (as in MMPY). Packing several fields into a wide word to save space may cause significant additional execution time costs.

The results of the NOVA benchmark are given in Table 5. In a sense, this program is the true test of the two machines, since it represents the type of program which certainly the MLP-900, and to a lesser extent the PDP-11/40E, were designed to run. For brevity, input/output instructions and interrupts were omitted from the NOVA benchmarks; in all other respects, the programs qualify as bonified emulators.

The NOVA is a 16-bit minicomputer, similar in flavor to the PDP-8 but with four general registers, more varied addressing modes, and a wider data path. As in the PDP-8, the ALU instructions only reference registers, and several low-level operations can be performed in a single ALU instruction (e.g., carry-bit setting, ALU operation, shifting, and skip-on-condition). The memory reference instructions are fairly typical. Finally, autoindexing of certain memory locations can occur within an indirect address chain. For more information on the NOVA computers, see [8].

The PDP-11/40E version of the NOVA emulator was adapted from an emulator written at CMU by Paul Drongowski. Several modifications and corrections were added. The programs for both machines were written quite painstakingly, with strong emphasis on speed. As Table 5 shows, the PDP-11/40E did consistently better than the MLP-900 on all the instructions. In the sample NOVA programs, the 40E was about 30% faster. Several reasons suggest themselves for this result. First, the PDP-11/40E might be expected to do well, since it's a 16-bit minicomputer emulating another 16-bit minicomputer. If the NOVA operated on only 12-bit words, for example, the ALU instructions on the 40E would probably experience a greater increase in overhead than on the MLP-900. Second, a reasonable amount of field extraction was necessary in the NOVA emlator, which raised the instruction-decoding overhead on the MLP-900. It is not clear, however, that this situation would be greatly different in emulating other machines. Third, the longer memory reference time on the MLP-900 contributed up to an extra microsecond over the PDP-11/40E version for some of the NOVA instructions (those requiring several accesses to memory).

It should be noted that the MLP-900 has the capability for greatly increasing its instruction-decoding speed of target instructions. This is done by using a changable "language board", which is essentially combinatorial circuitry tailored for a particular application (e.g., instruction decoding for a specific target machine). The MLP allows up to four such boards to be connected at one time, with programmed switching between them. Both the OE and CE have parts of their address space which serve as inputs to and outputs from the language board. Hence, tailor-made combinatorial circuitry could isolate all the important fields of a target instruction very quickly. Presumably, a language board would only be designed where the expense would be justified by doing a great deal of emulation of that specific machine.

Several conclusions can be reached from these benchmarks. While the MLP microinstructions seem to be higher level (capable of more complexity, in some sense) than the PDP-11/40E instructions, the latter are capable of significant data-path parallelism, which seems to substantially offset their primitiveness. The MLP instructions also are relatively expensive in time compared to the 40E instructions. The MLP's strong points are its large microstore and its data path width, though there is some reason to doubt that effective use of the wider word can always be made without incurring significant extra execution time costs.

V. Neutral Benchmark Comparison.

In addition to the straight performance benchmarks, an attempt was made to see how the two machines performed doing the same basic operation sequence, without appeal to their specific strengths or weaknesses. This is essentially a comparison of the basic architectures of the two machines. The most reasonable method of doing this seemed to be to compare "neutral" benchmarks (i.e., ones favoring neither machine), though it requires a decision on what constitutes a "neutral" program. In this case, programs were selected which (a) used primarily full-word operations, without field extraction, and (b) did not require a word wider than 16 bits. The basic unit of comparison was the number of instructions required to implement the desired operation sequence. Comparing execution times only has meaning if the final results of the two versions is the same. A double-word multiply, for example, is the same basic operation sequence, but the results are quite different on the two machines, due to the difference in data path widths.

Several programs were selected: ALLOC, the memory allocation benchmark

already introduced, and RLIST, a routine to reverse a circular list, were both considered fairly neutral. Also, the MMPY benchmark was modified to produce two-word, three-word, and four-word integer multiply routines (MMPY2, MMPY3, and MMPY4) on each machine. The comparison of program sizes is shown in Table 6. The execution time ratio for ALLOC and RLIST (the only programs where an execution time comparison was meaningful) was 0.88 and 0.86 respectively, assuming long list length -- that is, the PDP-11/40E was somewhat more than 10% faster on both programs.

The space comparison illustrates a couple of points. When the PDP-11/40E performed a task which did not require a large number of internal registers (e.g., ALLOC, RLIST, and MMPY2), then the number of instructions required was perhaps 10-20% more than on the MLP-900 (for "neutral" programs). However, when the registers were stretched thin (as in MMPY3 and MMPY4), then the PDP-11/40E version became significantly longer. The MLP-900, having many more registers than the PDP-11/40E, suffered considerably less on programs that made large temporary storage demands.

The main conclusion of this particular comparison is the following. For "neutral" progams which only require a small amount of intermediate storage, one can expect that each MLP-900 instruction will be worth about 1.1-1.2 PDP-11/40E instructions. The small amount of timing data on the neutral programs suggests that the 40E will execute these 1.2 instructions slightly faster than the MLP-900 will execute its one instruction. Obviously, this number of 1.2:1 will not be valid outside of the neutral/small-memory class of programs; however, it is still interesting as a base value in comparing the two machines, and perhaps useful in some cases as a point to extrapolate from.

VI. Comparison of Branch Costs.

Superficially, at least, a two-way conditional branch on the PDP-11/40E can be very costly in space and time. Consider the following example, which decrements R[0] by one and tests if it is zero:

```
D←R[0]-1;  R[0]←D;      Decrement R[0] by 1, restore into R[0]
SKIPZERO;               Test the condition "D equals zero".
NOOP;
SET
   GOTO LOOP;           Go to label LOOP if D≠0
   GOTO NEXT;           Go to label NEXT if D=0
TES
```

Note that each line is a new instruction, and the semicolons simply delimit the functions of a single microinstruction. SET and TES are not instructions but rather delimiters that tell the micro-assembler to place the enclosed instructions in consecutive locations, starting on an even boundary. When no explicit GOTO field occurs in an instruction, the assembler defaults the next-address field to the address of the next instruction in the source file. SKIPZERO is a mnemonic for a test-condition field which senses whether register D is equal to zero (in general, this cannot be done in the same instruction that loads the value to be tested into D).

The instructions "SKIPZERO", "NOOP", "GOTO LOOP", and "GOTO NEXT" are all capable of performing several other functions in the same instruction. Thus in this example, they are all essentially "place-holding" instructions, whose execution time is wasted. With care in programming, useful computation can usually be performed in all or most of the "place-holders". For example, the two GOTO instructions can be considered the first instructions of their respective control paths, instead of just the branches to the first instructions. Furthermore, the first one or two instructions of one of the diverging paths can frequently be pushed up into the NOOP and SKIPZERO

instructions, as long as this has no conflicting effect on the other possible path. In short, two-way conditional branches are usually not nearly as costly as one might initially think.

To support this claim, a count was made of all the unused instructions in two-way conditional branches for five of the PDP-11/40E benchmark programs. An unused instruction was defined as either a NOOP, or one that contained only a GOTO field or a SKIPZERO field (or similar condition-sensing specification). Out of a total of 19 two-way conditional branches, 9 unused instructions were found, or an average of about 0.5 unused instructions/branch. Since a place-holding instruction requires only a P1 clock pulse (140 nsec.), the expected wasted time/branch was 70 nsec.

These measurements were also done on early versions of the same programs. Interestingly, 20 unused instructions were found out of 18 conditional branches, with the expected cost thus being about one instruction and 140 nsec. This suggests that the cost of conditional branches may be considerably higher for certain situations, such as an early stage of program design or a non-proficient programmer.

The expected cost of a MLP-900 conditional branch instruction can also be calculated, though in a completely different way. On the MLP-900, a simple boolean function of two variables (i.e., flip/flops) can be evaluated in a single branch instruction. If the result is true, the jump address replaces the current instruction counter; if false, the next sequential instruction is executed. Since all branch instruction are executed by the control engine, the conditional branch can be overlapped if immediately preceeded by an OE instruction. Thus the cost of a conditional branch, if overlapped, is one instruction and 0 nsec. or if not overlapped, one instruction and 250 nsec.

The conditional branches out of five benchmark programs were examined. Out of a total of 16 conditional branches, 11 were overlapped with a preceeding OE instruction, or 70%. Hence, the expected cost of a conditional branch on the MLP-900 is one instruction and 0.3*250 nsec., or 75 nsec.

Thus, the PDP-11/40E conditional branches are, in practice, not as expensive as they appear. The expected costs of conditional branches on the two machines seem to be approximately equal, at least as far as the limited statistics collected indicate.

VII. Other Considerations: Ease of Programming.

One's initial impression about writing microcode for the PDP-11/40E is that it is fairly difficult to write good code. This impression is confirmed in practice. To achieve the performance that the 40E is capable of, one has to "think parallel" in programming it. This is complicated, however, by the inevitable hardware quirks which tend to break a unified scheme into a number of special cases. On the 40E, for example, selecting a general register for loading from the DMUX bus also causes the register's contents to be ORed onto the RD bus. Thus, the following instruction, an apparently perfectly valid example of a parallel data path operation, will perform incorrectly (unless R[1] is initially zero):

R[1]←D; BA←S; Load R[1] from D, and BA from Stack

There are more examples of such low-level hardware restrictions on the 40E, but the real difficulty in writing good microcode is the breadth of the solution space: many alternate ways of programming the same task exist (the intra-instruction parallelism essentially contributes another degree of freedom). Some of the alternatives are clearly inferior, but deciding which is the best or near-best frequently requires some sort of search technique.

One such technique is to see if the functions of an instruction can be moved up into preceeding instructions, thus allowing that instruction to be elimated. For example, consider the following microcode segment:

```
            S←D;
L1:         S←R[0];  GOTO COM;
            ...
L2:         D←R[1]+1;  R[1]←D;  GOTO COM;
            ...
COM:        B←R[0];
            BA←R[1];
            ...
```

Note that "S←. . ." means push a value onto the stack, while ". . .←S" means pop the top value off the stack. The object will be to eliminate one of the instructions starting at COM, if possible. Assume COM has only the two predecessors shown (L1 and L2). Both of these paths must incorporate the function(s) that are eliminated in the COM segment. Consider the function "B←R[0]", the first instruction in the COM portion. This can be added to the instruction at L1 without conflict, making "S,B←R[0]". However, "B←R[0]" can not be added to L2 because R[1] is aready specified. Hence, "B←R[0]" cannot be eliminated. So, consider the second instruction in the COM section, "BA←R[1]". We can move it over the "B←R[0]" instruction, since they are logically independent. This function cannot be overlapped with L1, but it can be overlapped with L1's predecessor, making "S←D; BA←R[1]". Since there are not data dependencies or conflicts, the same effect is achieved. In addition, "BA←R[1]" can be overlapped with L2, thus we can eliminate the "BA←R[1]" instruction entirely. The resulting code segment looks like:

```
            S←D;  BA←R[1];
L1:         S←R[0];  GOTO COM;
            ...
L2:         BA,D←R[1]+1;  R[1]←D;  GOTO COM
            ...
COM:        B←R[0];
            ...
```

This has saved one instruction and 140 nsec. on both control paths. In fact, we can save an additional 140 nsec. on only the L1 path by incorporating the "B←R[0]" function in the instruction at L1 and then branching to the successor of COM. This does not affect the time on the L2 path, however.

One's initial impression about the MLP-900 is that it is considerably easier to program than the PDP-11/40E. This is undoubtedly true if one is not concerned with achieving maximum performance. If very high performance is a goal, however, then the issue is not so clear. Maximum parallelism on the MLP-900 occurs with an even mix of OE and CE instructions. The objective is thus to maximize the OE/CE pairing while minimizing the overall number of instructions (or pairs). This requires not only selecting the appropiate microinstructions to carry out the desired operation, but also determining the best distribution of data within the two engines to facilitate selection of the appropiate microinstructions.

To illustrate the problem of data distribution, consider the storage of a target machine instruction in the MLP-900. Parts of a target-machine instruction take place in arithmetic operations (which is an OE function), and other parts are used in bit testing and case statement indexing (which are both CE functions). Storing the instruction in only one engine would hinder access by the other engine to its necessary data, so a solution is to keep a copy of the target instruction in both engines. Sometimes, however, data required by the CE must first be operated on by the OE, so other complications arise.

The point is that programming the MLP-900 for optimum speed can be surprisingly time-consuming, since other factors can have a pronounced effect on the speed besides simply the algorithm selected.

VIII. Conclusions.

For the two timed "neutral" benchmarks (ALLOC and RLIST), the PDP-11/40E appeared to be 10-15% faster than the MLP-900. For the benchmarks which included more byte-handling and field extraction (TRANS and NOVA), the 40E was 25-30% faster. The MLP-900, on the other hand, was over four times faster than the 40E for the data-path benchmark (MMPY). This last result reflects the fact that the 40E had to perform roughly four times as many operations for the same result as the MLP-900.

The choice of which machine to use may come down to deciding whether the MLP-900's two strong points, a 36-bit data path and 4K microstore, are necessary to the application. The two machines seem fairly equivalent in speed (except on wide data-path applications). Thus, if one is primarily interested in emulation of machines with 16-bit words or less, then the PDP-11/40E is probably preferred. (The 40E also has the advantage that the console switches can be programmed to emulate the target machine's console for hands-on debugging and operation.)

On the other hand, if particularly large machines are to be emulated, especially if they have a wide word-length, then the MLP-900 is probably preferred. The alternative for the 40E might be to program frequently-needed primitives in microcode (such as multi-word operations to simulate a wider data path), while using regular PDP-11 machine code for the more mundane parts of the simulation. This would be quite slow compared to the MLP-900, of course.

There are a couple of other external factors which might influence a choice between the two machines. First, if the application will involve automatic compilation of microcode (as eventually intended in the SMCD project), then ease of producing a compiler to generate highly-optimized microcode would certainly be a factor. For

either machine the task would be an extremely difficult one; a subjective impression, however, is that the PDP-11/40E would be slightly easier than the MLP-900 because of the more regular manner in which data is handled, and the more straightforward optimization goals. On the MLP-900, both these aspects seem much more ill-defined.

The second external factor is accessibility, a more pragmatic topic. The extensions to the PDP-11/40E, while conceived and executed as a research project at CMU, hold the possibility of eventually being offered as an option by Digital Equipment Corporation on future PDP-11/40's. The MLP-900, though originally a prototype machine that never reached production, is a usable system at ISI and has fairly wide availability via the ARPA Network. Thus, each machine has its own type of accessibility which will be appropiate to different types of users.

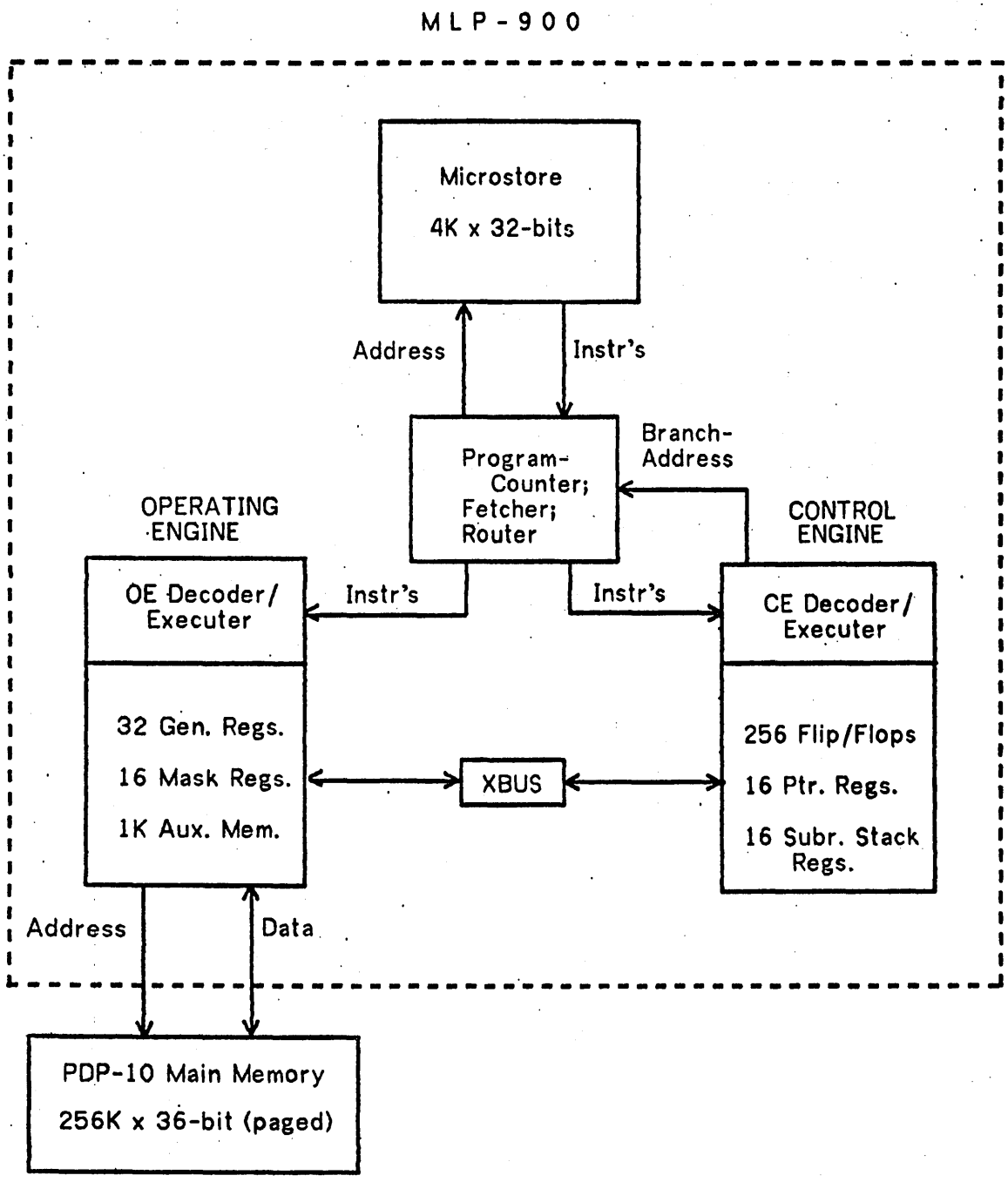

Figure 1.

Schematic Diagram of MLP-900 Components.

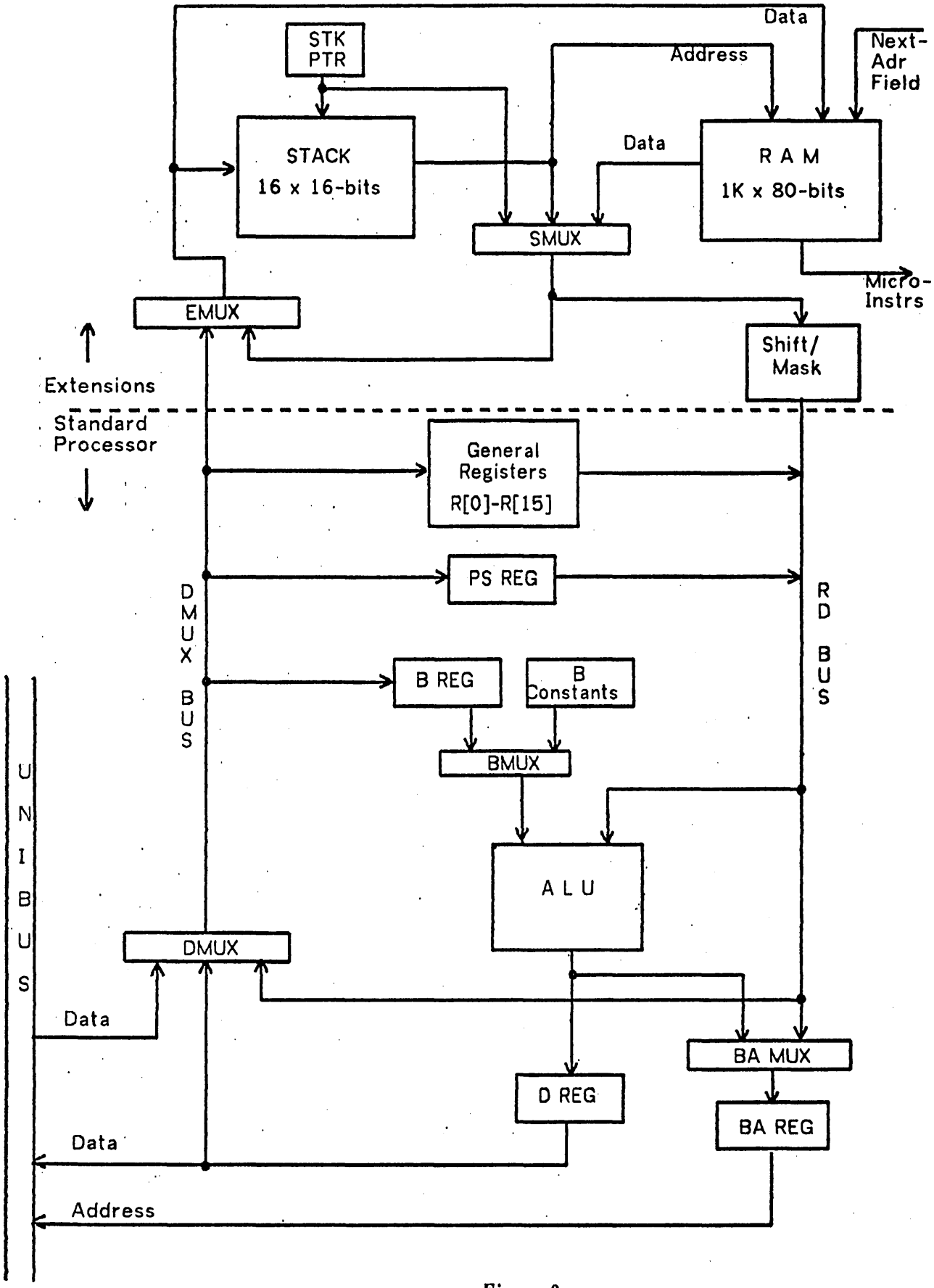

Figure 2.

PDP-11/40E Data Path Diagram.

Table 1.

Major Parameters of the PDP-11/40E and MLP-900 Microprocessors.

Parameter	PDP-11/40E	MLP-900
Year Designed	1970 (40E: 1973)	1968
Data Path Width	16 bits	36 bits
Basic Cycle Times	140, 200, 300 nsec.	250 nsec.
Microstore Size	1K x 80-bits	4K x 32-bits
Bits/Instruction	80	32
Total Time for Memory Reference (including microinstruction times)	780-1180 nsec.	1300 nsec.

Table 2.

MMPY: 64-Bit Integer Multiply Benchmark

Measure	PDP-11/40E	MLP-900	Ratio: 40E/MLP
# Instructions (% avail ustore)	85 (8.3%)	32 (0.8%)	2.7
Execution time (usec)	421.2+5.2N	108.4+.75N	
Average exec. time (usec)	587.6	132.4	4.4
Maximum exec. time (usec)	754.0	156.4	4.8

N = number of one-bits in multiplier.

Table 3.

TRANS: Character Translation Benchmark
(Two characters/word)

Measure	PDP-11/40E	MLP-900	Ratio: 40E/MLP
# Instructions (% avail ustore)	26 (2.5%)	25 (0.6%)	1.04
Execution time (usec)	.88+3.2N*	.75+4.23N*	0.76 (large N)

N = # characters in string.
* N assumed even

Table 4.

ALLOC: Memory Allocation Benchmark
(Boundary-tag method)

Measure	PDP-11/40E	MLP-900	Ratio: 40E/MLP
# Instructions (% avail ustore)	52 (5.1%)	51 (1.2%)	1.02
Avg. execution time (usec)	11.24+3.4N	13.0+3.85N	0.88 (large N)

N = # of blocks in list checked unsuccessfully.

Table 5.

NOVA: Minicomputer Emulator Benchmark

Measure	PDP-11/40E	MLP-900	Ratio: 40E/MLP
# Instructions (% avail ustore)	169 (16.5%)	259 (6.3%)	0.65
Execution times - Memory reference instructions (usec)			
LDA:	3.96	4.85	0.82
STA:	3.84	4.85	0.79
ISZ, DSZ:	5.08*	5.90	0.86
JMP:	2.86	3.80	0.75
JSR:	3.34	4.05	0.82
If PC-relative, add:	0.30	0.50	
If AC-relative, add:	0	0.50	
If Indirect Address, add:	2.30	1.80	
If Autoindexing, add:	0.90	1.55	
* If skip occurs, add:	0.16	0	
Execution times - ALU instructions (usec)			
COM, MOV, INC, ADD, AND:	4.32	6.50	0.66
NEG, SUB:	4.62	6.50	0.71
ADC:	4.62	9.00	0.51
If NOLOAD, subtract:	0.32	0.25	
If L shift, add:	0.46	0.25	
If R shift, add:	1.00	0.75	
If Swap-bytes, add:	0.10	0.50	
If Skip-field specified, add:	0.28**	0**	
** If skip occurs, add:	0.30	0.25	
Execution times - Sample NOVA programs (usec)***			
Sum-of-integers (1-20):	249.8	356.2	0.70
16-bit integer multiply:	310.9	424.2	0.73

Note: Variation in the memory reference time of the PDP-11/40E may cause the execution times for the individual instructions to vary a few hundred nanoseconds depending on the instruction mix being executed. The timings of the sample NOVA programs, however, take this into account.

*** The slowest NOVA computer performs Sum and Multiply in 293.4 and 334.3 usec., respectively. The fastest NOVA (800 Series) performs them in 50.6 and 57.8 usec.

Table 6.

Neutral Benchmark Comparison.
(Comparison of # Instructions)

Program	PDP-11/40E	MLP-900	Ratio: 40E/MLP
ALLOC: Memory allocation	52	51	1.02
RLIST: List reverser	16	14	1.14
MMPY2: Two-word multiply	38	32	1.19
MMPY3: Three-word multiply	71	39	1.82
MMPY4: Four-word multiply	85	44	1.93

APPENDIX A.

MLP-900 Microinstruction Examples.

The following examples should give an idea of the types of functions that MLP-900 microinstructions can perform. R.i are 36-bit general arithmetic registers in the OE, M.i are 36-bit mask registers in the OE, P.i are 8-bit pointer registers in the CE, and F.i are flip/flops in the CE.

(1) *R.1←R.1+R.2 (M.0);* (GEAR instr)

Add the contents of R.2 to R.1 using only the masked-in bits specified in mask register M.0. The round parentheses around "M.0" indicate that the masked-out bits in R.1 are left unchanged after the store -- i.e., only the masked-in bits of R.1 are replaced.

(2) *R.0←R.0 AND 377 \10 [M.1];* (GEAR instr)

AND R.0 with the octal literal 377 (using mask register M.1), and then shift the result left 10 (octal) bits. The square brackets around "M.1" indicate that the entire result is to be stored back into R.1.

(3) *R.37←R.37 OR @P.1;* (GEAR instr)

P.1 should have a value between 0 and 37, pointing to a general register. The effect is to OR R.37 with the general register indicated by P.1, and replace the result into R.37. The default mask register of M.0 is used.

(4) *IF F.100 OR NOT F.101 GOTO L1;* (BRAT instr)

Test the two flip/flops F.100 and F.101. Under the logical operation shown, branch to label L1 if the result is true. L1 must be within -177 to +200 (octal) instructions away from the current instruction. There is also a branch instruction (BEAD) that allows a full absolute address into control memory.

(5) *P.4←P.4-1, IF NOT F.47 GOTO L2;* (BRAD instr)

This is a single instruction that increments/decrements a pointer

register by a small literal, and simultaneously performs a test on a flip/flop. There are a number of flip/flops whose function is to monitor an assigned pointer register and be set true only when the pointer register contains a specific value (e.g., 0, -1, etc.). Hence this instruction can readily be used for loop control.

(6) *FOP R.0+1; WOP R.7;* (CEDE instrs)

These are two CEDE instructions. The FOP does an implicit "R.0←R.0+1" (where "1" can be replaced by any small number or a general register name), and then starts a memory fetch from the virtual address in R.0. One or more instructions later, a WOP is done, which waits for the data word, and loads it into R.7 when it is available.

(7) The following is a simple program to calculate the sum of integers from 1 to 100 (decimal).

```
        R.0←145;                        Initialize counter to 101
        R.1←0;                          Initialize sum to 0
LOOP:   R.0←R.0-1;                      Decrement counter by 1
        R.1←R.1+R.0;                    (OE/
        IF NOT ZRF.1 GOTO LOOP;           CE pair)
        RETURN;
```

ZRF.1 is the name of one of the status flip/flops implicitly set by a GEAR or SHIN instruction. It has the value true if the last GEAR operation (though <u>not</u> the one in the current cycle) produced a zero result. In the program above, there is an OE/CE pair which is executed in one cycle, hence the ZRF.1 usage refers to the count-decrement operation at LOOP.

An improved version of this program is shown below. This version uses a pointer register to hold the count, so only counts up to 255 can be used. Note that pointer registers can take part in a GEAR operation, even though they are part of the CE.

```
        R.1←0;                                  (OE/
        P.0←144;                                  CE pair)
LOOP:   R.1←R.1+P.0;                            (OE/
        P.0←P.0-1, IF NOT ZSI.0 GOTO LOOP;        CE pair)
        RETURN;
```

ZSI.0 is a flip/flop that is true when P.0 contains zero. Note that the pre-modification value of P.0 is used for the addition to R.1 and the ZSI.0 flip/flop. Hence one additional pass through the loop occurs, adding zero to R.1. This version executes about twice as fast as the previous version, since the loop is only one cycle long instead of two.

APPENDIX B.

PDP-11/40E Microinstruction Examples.

To clarify the text description, here are some simple examples of some PDP-11/40E microinstructions. (Note: the semicolons delimit functions within a single microinstruction, not microinstructions themselves.)

(1) *BA,D←R[0]+1;*

Shorthand for:
Select R[0] as source for RD bus.
Select Constants as source for BMUX.
Select the constant 1 from constants.
Initiate ALU operation of addition.
Clock ALU output into D and BA registers on a P2 pulse.

A P2 pulse is required, since D can be loaded only by a P2 pulse. BA is loaded at the same time. If B (for example) were also selected for loading, a P3 pulse would also be necessary, thus implying CLK=3 and an execution time of 300 nsec (instead of 200 in this example).

(2) *S,B←R[2];*

Shorthand for:
Select R[2] as source for RD bus.
Select RD bus as source for DMUX.
Clock DMUX into B reg on a P1 pulse.
Push value on DMUX bus onto stack (S) at end of instruction execution.

Since only the B register is selected, a P1 pulse is required. A P3 pulse would also work, but since D is not selected, there is no point in requiring 300 nsec instead of 140 nsec, hence CLK=1 in this instruction.

(3) *D←R[1]-B; R[1]←D;*

Shorthand for:
Select R[1] as source for RD bus.
Select B reg as source for BMUX.
Select D reg as source for DMUX bus.
Initiate ALU operation of subtraction.
Clock ALU output into D register on P2 pulse.
Clock DMUX bus into R[1] on P3 pulse.

Only one general register can be selected in any microinstruction. Since R[1] is selected as input to the ALU, only R[1] can be selected to be loaded also. Note that R[1] can be clocked on either a P1 or a P3 pulse. However, P2 is already required in this microinstruction (to load D), and there is no P1-P2 pulse sequence, so P2-P3 (i.e., CLK=3) is required. The net effect is to decrement R[1] by the contents of B.

(4) Here is a simple program to calculate the sum of integers from 1 to 100 (decimal):

```
        D←144;  B←D;          Initialize B to 100 (decimal)
        D←0;  R[1]←D;         Initialize sum
LOOP:   D←R[1]+B;  R[1]←D;    Increment sum by count
        D←B-1;  B←D;          Decrement count by 1
        SKIPZERO;             Test if D is 0
        NOOP;                 (Place-holder)
        SET
          GOTO LOOP;          Continue if D≠0
          EXIT;               Done if D=0
        TES
```

The SET and TES are not microinstructions, but rather delimiters which tell the micro-assembler to place the enclosed instructions on an even (in this example) boundary. The SKIPZERO function tests if register D equals zero. If true, then the EXIT instruction is executed.

By a realization of two facts, the size and execution time of this program can be reduced. First, "GOTO LOOP" can be the first instruction of the new control path, rather than just a place-holder. Thus, we can combine the first instruction of the loop with "GOTO LOOP" in the SET/TES pair, and simply branch to this instruction after initialization. Second, by realizing that an extra add to the sum is harmless after the count is decremented to zero, the loop can be rolled up another instruction, with the following equivalent program (which is 25% cheaper in both space and time).

```
        D←144;  B←D;
        D←0;  R[1]←D;
LOOP:   SKIPZERO;
INIT:   D←R[1]+B;  R[1]←D;
        SET
          D←B-1;  B←D;  GOTO LOOP;
          EXIT;
        TES
```

This example illustrates the sort of low-level optimization that goes on when programming a horizontally-encoded microprocessor.

REFERENCES

[1] W. A. Wulf, "ALPHARD: Toward a Language to Support Structured Programs," Carnegie-Mellon University Technical Report, April 1974.

[2] M. R. Barbacci and D. P. Siewiorek, "Some Aspects of the Symbolic Manipulation of Computer Descriptions," Carnegie-Mellon University Technical Report, July 1974. Also appears in: ***Second Workshop on Computer Descriptions Languages***, Darmstadt, W. Germany, July 1974.

[3] H. W. Lawson, Jr. and B. K. Smith, "Functional Characteristics of a Multi-Lingual Processor," ***IEEE Trans. Comput.***, vol. C-20, July 1971, pp. 732-742.

[4] L. Gallenson, J. Goldberg, R. Mayson, D. Oestreicher, and L. Richardson, "PRIM User's Manual," University of Southern California, Information Sciences Institute Technical Report ISI/TM-75-1, April 1975.

[5] S. Fuller ***et al.***, "PDP-11/40E Reference Manual," Carnegie-Mellon University Technical report, June 1975.

[6] W. A. Wulf and C. G. Bell, "C.mmp -- A Multi-Mini-Processor," in ***1972 Fall Joint Comput. Conf.***, vol. 41, AFIPS Press, 1972, pp. 765-777.

[7] D. E. Knuth, ***Fundamental Algorithms***, Menlo Park, Calif.: Addison-Wesley Pub. Co., 1968, pp. 441-442.

[8] ***How To Use the NOVA Computers***, Data General Corp., 1972.