Computing Science Technical Report #80

A 32-Bit Processor Design

Stephen C. Johnson

Bell Laboratories
Murray Hill, New Jersey 07974

April 2, 1979

A 32-Bit Processor Design

Stephen C. Johnson

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper describes a user-level instruction set for a 32-bit processor. The machine is simple, compact, and well suited to the C language. In fact, even for 16-bit applications it is reasonable to expect the instruction space for C programs to be 10 to 15 % smaller than for the PDP-11.

Most of the desirable properties of this instruction set are the direct result of the design methodology:

1. A design was constructed.
2. A C compiler, based on the portable C compiler, was constructed to compile C for the current machine.
3. Measurements were taken to investigate the effectiveness of the instructions and addressing modes in the design. The measurements, based on real C programs, led to redesign, and a return to step 1.

This methodology led to a machine design which seems exceptionally attractive to at least one software person (the author).

April 2, 1979

A 32-Bit Processor Design

Stephen C. Johnson

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction and Methodology

This note proposes, in considerable detail, a 32-bit machine design. It is greatly influenced by a 16-bit machine proposal of A. G. Fraser. More important than the design itself is the methodology used to design the machine. At every stage in the evolution of this design, a C compiler was constructed, using the portable C compiler as a base. Measurements were then taken on a large body of C programs. Based on these measurements, the machine design was adjusted, the compiler adjusted, more numbers collected, etc. After several iterations, the design converged to the one in this proposal. The final design has several properties:

1. Although it is a 32-bit machine, C programs tend to be smaller than on the PDP-11. Since modern technologies often lead to memory-limited architectures, this fact, together with the relative simplicity of the design, suggest that the machine can be quite fast.
2. The machine is simple; there are no complicated address modes. Fault recovery should be straightforward.
3. With the possible exception of the call, save, and return instructions, the machine does not seem abnormally slanted towards the C language.
4. The structure of the machine allows a number of "fallback" machines to be built in the event that the entire design cannot be packaged on a board or a chip. The methodology allows a close estimate of the cost in performance of these various fallback positions.

This fourth point is worth some additional elaboration. The simplest machine proposed has only three address modes; these correspond to memory access, immediate access to constants, and register access. The C compiler generates code for this simple model. If this simplest machine were built, something over twice as much memory would be required to hold the program; the control, however, would be far simpler than the final proposal.

If one examines the behavior of real programs when they are compiled for the basic machine, it turns out that most of the "action" takes place in a small corner of the instruction set. For example, while the basic machine is able to load full 32-bit immediate constants into registers, the values 0 and 1 are extremely common while numbers around a million are quite rare. If the machine is provided with an instruction which loads small immediate constants directly from the instruction, the 32-bits required to hold the immediate constant will be saved in a substantial fraction of the moves. Thus, the strategy is to propose a collection of optimizations, and try them on real C programs.

One cannot add optimizations to the instruction set forever. For one thing, the opcode space becomes exhausted. One way of dealing with this constraint is to apply the optimizations only to those instructions where there would be a substantial saving in time or space. Unfortunately, unrestrained use of this principle can cause the control for the CPU to become quite complicated (although the rest of the machine may remain quite simple). This is especially true with LSI, which, with today's technology, places a heavy penalty on irregularities in the opcode set.

The remainder of this document will set forth a design, and then attempt a justification of this design by examining its performance on a large body of C code. An appendix suggests a bit layout for the instructions.

The Basic Design

The machine will have a number of 32-bit registers:

PC The program counter

PSW The program status word

SP The stack pointer

Z A "register" always containing zero.

R2-7 Six symmetric general-purpose registers.

For C purposes, R2 and R3 will be used as scratch registers, while R4-R7 will be used for register variables.

The machine has three basic addressing modes.

X 32-bit immediate constant.

R One of the six general purpose registers, SP, or Z.

X(R) 32-bit offset from one of the six general purpose registers, SP, or Z. Note that an offset from Z gives a general static memory location.

These address modes represent the basic, "compiler's eye" view of the machine. Notice that all of these address modes can be used as source operands, and all but X and register Z can be used as destinations as well.

The machine will also have optimizations of these address modes; the intent is to represent the common cases of the above address modes with fewer bits of instruction space. For example, 16-bit immediate constants, 16-bit offsets from registers, constants in the range 0-15, and common offsets from the stack pointer are all desirable optimizations. This topic will be discussed at length later.

All the operations which take place in registers will be done to full 32-bit width, and the condition codes will reflect the 32-bit value. All memory references will have a width associated with them, which will be either:

C Char (8 bits)

S Short (16 bits)

L Long (32-bits), the "int" for the C compiler.

Floating-point will be explicitly ignored throughout the machine discussion to follow.

We shall make some basic assumptions about the machine organization, primarily that the instructions are organized as 16-bit quantities. Each opcode will be read, and then, as necessary, one or more 16-bit quantities following it will be read to provide for the immediate data and offsets needed in the instructions. The penalties of doing this, and the advantages, will be discussed in a later section. The memory interface will use a modification of a simple idea proposed by David Wheeler of the University of Cambridge; since this idea apparently has not appeared in print, the next section will discuss it. Following sections discuss the opcodes, and their optimization.

The Wheeler Device

The design of an opcode encoding is complicated by the need to specify operand width for memory references. The effective operand width for registers and immediates is taken to be 32 bits by default. David Wheeler has suggested a scheme that gives up one bit of address space, and uses a simple encoding to specify the operand width. The advantages of this scheme in regularizing the instruction set will become evident in later sections.

It is assumed that halfwords (16-bit quantities) will only be accessed from even-numbered byte addresses, and fullwords (32-bit quantities) will only be accessed from byte addresses divisible by four. This implies that, out of the twelve possible combinations formed by the low-order two bits of the address and the three possible widths, only seven combinations are legal: these can be encoded in three bits.

Since all memory accesses in the current machine have the form X(R), it is particularly

convenient to encode the width only in the low-order three bits of the offset, X. Thus, all pointers and addresses are byte addresses. If X has the bitwise representation

xxxx. . . .xxxyyy

then the byte address for the reference is taken to be

xxxx. . . .xxxzz + R

and the width is taken to be *w*, where *zz* and *w* are given from *yyy* by the following table:

yyy	*zz*	*w*	
000	**	*	error
001	00	1	
010	00	2	
011	01	1	
100	00	4	
101	10	1	
110	10	2	
111	11	1	

This scheme will allow pointers in the machine to be represented as byte addresses, as with many of the existing C compilers. The table-look-up implied in the above can be easily done by microcode or by hard-wired logic.

By default, following the C conventions, byte accesses will not sign-extend, and halfword accesses will sign-extend. It would be straightforward to steal another bit from the offset to encode more control over sign-extension.

The Instruction Set

This section will discuss the basic philosophy, and leave the optimizations until later. For each opcode, the address modes which are acceptable for its source and destination will be discussed.

The conditional branch has a 4-bit condition selector, capable of selecting the 6 arithmetic relationals, the 4 logical relationals, and unconditional branch, at a minimum. The address has the general form X(R), and must specify a halfword address.

The most frequent ops exist in a general form with arbitrary source and destination. These opcodes include **add**, **subtract**, **and**, **or**, **exclusive or**, **compare**, **left** and **right shift**, **insert under mask**, and **load address**.

The other operators may describe an arbitrary source, but will have the destination field restricted to a register. These operators include **multiply**, **quotient**, and **remainder** (both **signed** and **unsigned**), and **call**, **save**, and **return**.

Call, Save, and Return

The stack frame will grow downwards in memory. Note that this puts some additional demands on the memory management unit, but it will permit a lot of optimization of the use of the Stack Pointer.

The stack frame for a function looks like

```
            Incoming Argument n
            Incoming Argument n−1
            . . .
old SP→     Incoming Argument 1
            old PC save area
            save area for register variables
            . . .
            automatic storage
            . . .
            temporary locations used for expressions
            . . .
            Outgoing Argument m
            . . .
SP→         Outgoing Argument 1
```

The **call** opcode will push the return PC onto the stack, and go to the destination address (form X(R)).

The **save** opcode will take a register specifier, R, and a general source. Registers R through 7 will be pushed onto the stack, and the SP will be decremented by four times the source value.

The **restore** opcode has the same arguments as **save**; it increments SP by four times the 16-bit immediate quantity, reloads the registers, and reloads and branches to the saved PC value.

In both cases, the R values must be encoded so that it is possible to do a **save** or **restore** without storing any registers. This should be possible, since there is little point in saving Z. Note that the registers must be reloaded in the reverse order that they are stored in.

Other Opcodes

There are other opcodes which would be desirable. Negating and taking the ones complement are desirable operations; they will be permitted with register source only. The ability to sign-extend from the lower half of a register is also useful.

The left and right rotate operators are an anachronism; they are not required by most high-level languages, including C, and rarely arise even at a lower level; they will not be included.

A "load address" instruction, computing the byte address of a memory reference, is an attractive "3-address" form of add. This is used substantially in C code, and would be even more useful in languages (such as FORTRAN) that pass arguments to procedures by reference.

System Call, Floating point, etc.

It will be necessary to supply a number of extensions to the basic machine design, in order to do things such as system calls, return from interrupts, etc. These may be provided in many different ways, whose discussion is beyond the scope to this paper.

Optimizations

From the software designer's point of view, the machine should have a regularity of structure; however, since real programs do not exploit all instructions equally, it will be desirable to provide some optimizations in the basic design. The key criterion for any of these optimizations is that they should provide a special case of some more general construction; the compiler will generate code for the general case, while the assembler will choose the instruction form which most efficiently gives rise to the intended effect.

The following optimizations are almost certainly be desirable for some opcodes.

x(R) The X(R) addressing mode should have an optional form, x(R), where x is a 16-bit unsigned offset. This should probably be legal anywhere X(R) is, since many programs of interest will be able to fit in 64K bytes. Since I and D space will be separated, this will allow for

optimized references to data space.

x The X immediate opcode should allow for a 16-bit signed constant, x, to be used instead. (The question of whether x is signed or unsigned is not too important, since integers in the range of 0 through 100 or so make up the major use of such constants). Many opcodes, especially **move**, **compare**, **add**, and **subtract**, frequently have constant operands, and many, if not most, of these constants are less than 32,000.

c Carrying this a step further, it would be nice to have the ability within an opcode to reference the immediate constants 0 through 15. The constants 1 through 16 would even be better. In some cases (for example, **shift**), 1 through 32 would possibly be nice. Short immediate constants seem particularly attractive for **add** and **subtract**. They are also probably desirable for **save** and **restore**.

n(SP) There are many references to the current stack frame, and the current stack frame, on the average, is not too large. The ability to access the first few integers on the stack is exceptionally desirable. In particular, the ability to get at the first 16 would allow nearly all argument pushes, accesses to temporaries, and automatic variable references to be optimized, and, in many functions, many of the accesses to incoming arguments as well.

These optimizations do not change the basic addressing structure of the machine. They permit the assembler to encode certain common instructions to save space. By applying the methodology to this design, it should be possible to closely estimate how much can be saved from applying each optimization to each opcode. This saving can then be weighed against a potential cost in hardware and/or opcode decoding complexity.

Faults and Interrupts

When there is an interrupt, it should be possible to completely save the status of the executing user; the status should be restored when the user is to be restarted. This is most easily done by prohibiting interrupts except between instructions.

In the processor, an interrupt, fault, or system call should look like a function call, except that the previous values of the SP and PSW should be pushed onto the stack along with the PC. The return from interrupt should undo this operation.

The interrupt vector locations consist of several 32-bit words: these have the new PC, new PSW, and new SP. This implies that there are multiple stacks.

Unfortunately, faults may happen in the middle of instructions. The key difficulty is to be able to find the start of the instruction which faulted. Ideally, the PC returned after a fault should be the PC of the beginning of the instruction. This would require the PC value to be saved at the start of every instruction. Alternatively, a field in the PSW could be defined which is set to 0 at the start of every instruction, and incremented by 1 every time the PC is incremented. Using this field and the value of the PC at the time of the fault, the start of the instruction can be found, and, if desired, the instruction restarted. Note that in both cases, special action must be taken to ensure that the location from which an illegal branch or call is taken may be recovered.

The faults that will be provided by the machine are

1. Division by 0
2. Illegal Memory Access (bad alignment or no permissions to access).
3. Illegal Opcode

We propose having no trap on overflow, and no checking for overflow on field insertion. The job is not easy to do, in the first place; there are also some very ambiguous cases where it is not clear whether a trap should be sprung (such as in left shift). Address arithmetic, being done in unsigned mode, might cause unanticipated overflow traps (this is a serious problem on the Honeywell 6000, but need not be too bad in a 32-bit machine if care is taken). Moreover, faulting on overflow makes operations like addition non-associative, making the behavior of programs dependent on the whim of the compiler writer.

Justification for the Design.

As discussed in the introduction, during the course of the design methodology much data was collected about the behavior of the machine when used as a target for a C compiler. The data primarily represented static measurements, which reflect the space utilization of the machine. The time utilization is difficult to measure and difficult to intuit theoretically, so it will be regretfully ignored in this section.

The basic source of the data was a large body of utility programs, written in C. The validity of the numbers was strengthened by measuring other programs, including two operating systems, a Pascal compiler, and several applications programs. The numbers were remarkably consistent, and, while the details differ slightly, the conclusions apply to all the programs examined.

Perhaps the most striking feature of the proposed design is that there are fewer registers than most 32-bit machines proposed or on the market. Indeed, the number of registers is a critical parameter. If a machine has too many registers:

1. Hardware is wasted providing the registers, and selecting them.
2. Time is wasted saving and reloading them across function calls and context switches.
3. Opcode space is lost. As the number of registers increases, the opcode space needed to reference two registers increases quadratically. For example, with 8 registers there are 64 register-to-register move instructions, while with 16 registers there must be four times as many: 256.

If there are too few registers, it seems evident that the machine will have to do many stores and loads of temporaries in order to get its work done. Two scratch registers seems quite sufficient for the machine as constituted; a third scratch register would save at most one percent in the program size.

As far as the number of register variables is concerned, the machine actually has one more than the PDP-11; there is no hard evidence that this is "enough". The existence of a large body of programs with exactly three "register" declarations per function makes unbiased statistics hard to collect.

The utility of 16-bit offsets from registers and 16-bit immediate constants is considerable. Although this machine should be able to deal with full 32-bit addresses and constants, most processes will probably be small. Even for large processes, organizing the data area so that the frequently referenced data lies at the lower addresses should allow these optimizations to be applied as well. For the immediates, not only are many 16 bits, but most are quite small (under a hundred). This leads to considerable saving for some instructions by making a short (4 bit) immediate constant mode.

When the sample was compiled, over 62,000 instructions were generated, taking up 194,000 bytes, for an average of 3.1 bytes per instruction. If there had only been the three most general address modes, the program would have taken 2.3 times as much space; of the saving, 45% comes from having 16-bit offsets, 19% by having 16-bit immediates, 15% by being able to easily access stack elements, 12% by having short (4-bit) immediate constants, and 9% by being able to have a short (PC-relative) form of branch. The PDP-11 compiler produces program sizes of 230,000 bytes (unoptimized) or 215,000 bytes (optimized) for the same sample. Thus, on the PDP-11, the optimizer saves about 7% in space over the unoptimized PDP-11 version; the (unoptimized) code for the current machine is about 10% smaller than the *optimized* PDP-11 version. Additional improvement could be expected if a peephole optimizer were applied to the new machine as well.

These figures depend on the assumption that both instruction and data addresses are 15 bits or less. Various kinds of memory mapping, for example lack of separated instruction and data space, may force one or another of these address forms always to be 32 bits. For the sample, if instruction addresses became 32 bits, the size of the program would increase by about 8%, while if data addresses were 32 bits the program size would grow by about 16%. This implies that having a mapping unit which can separate instruction and data spaces is compacts code considerably.

The addressing optimization which permits easy access to the first 16 integers on the stack is surprisingly powerful. These references represent arguments for called functions, temporary

references, automatic variables, and, for small functions, references to incoming arguments as well. In the sample, about 90% of all references off of the SP are references to the first 16 words of the stack frame. Roughly 41% of all opcodes generated were moves (46% of which were moves of arguments onto the stack to prepare for calls), 19% were branches, 16% were call/save/return operations, 10% were plus, minus, and load address, 10% compares, and 4% for all other ops.

Summary and Discussion

The machine described above is a relatively simple, 32-bit machine; C program sizes appear to be at least 10% smaller than on the PDP-11. It is reasonable to assume that the machine could be made to run quickly, since the space saving is not attained by strange encodings or other dirty tricks. Its simplicity should allow self checking and fault recovery to be done easily.

A great deal of the credit for the design should be given to the methodology. A C compiler was designed at each stage; the current design was evaluated on a body of C code, and then the design was changed. The current design represents over a dozen iterations, starting from a 16-bit machine.

It is necessary to keep in mind the limitations of this methodology. If, say, a dozen factors need to be considered in a machine design, and two or three of them can be objectively measured while the others cannot, there will be a natural tendency for the factors which can be measured to dominate the discussion of the design. The measured factors may be optimized at the expense of creating extreme difficulties elsewhere in the design.

In the current study, program space was studied, under the assumption that it was correlated with execution time. This assumption may fail to be productive if the machine being considered uses any of a number of common hardware techniques (caches, pipelines, separate instruction and data busses, wide busses to name just a few). The dynamic behavior of programs, and measurements for languages other than C, would clearly be worth further study. Nevertheless, this machine has some known, objectively measured, advantages. The methodology gives a framework for further iteration and comparison with other designs. While this machine may never be built, the methodology will almost certainly be used again.

An Appendix discusses the bit layout. A second contains some discussion about 3-address machines.

Appendix I: Bit Layout

The bit layout of the instructions seems to have a considerable effect on the ease of LSI representation of the machine; for microcoded machines, the bit layout is far less critical. This section gives a possible bit layout, whose main purpose is to demonstrate that a machine with the optimizations discussed above (and a few others) can be encoded easily in a 16-bit opcode.

The main issues in the bit encoding seem to be:

1. The boolean functions needed to select the sources, destinations, and opcodes from the input word should be as simple as possible.
2. There is some advantage in having immediates arranged in the instruction word so that they can be gated directly onto the bus, without any need for shifting.
3. Having source and destination addressing share bit layout, and thus circuitry, seems desirable as well.
4. As much as possible, the opcode set should be complete, since (especially in LSI) checking for exceptions is expensive and wasteful.

The key to the present encoding is the description of the source and destination in 6 bits:

uu rrr x

These bits will be interpreted as follows:

00	*rrr*	0	Register *rrr*
00	000	1	16-bit immediate constant
00	001	1	32-bit immediate constant
01	*rrr*	*x*	*rrrx*, immediate value
10	*rrr*	0	16-bit offset from *rrr*
10	*rrr*	1	32-bit offset from *rrr*
11	*rrr*	*x*	(*rrrx*00 + SP), integer width

This encoding is laid out with several objectives in mind. Those sources which are also destinations correspond to the bit patterns

1* *** *

and

00 *** 0

(The first is all memory references, the second registers). The bit pattern

00 000 0

refers to the Z register, however, and cannot be a destination.

There are three instruction forms, corresponding to branches, opcodes whose destination is memory, and opcodes whose destination is a register.

Branches

The PC-relative form of the branch operator is represented by the 16 bits:

000 *s ccc xxxxxxx y*

The *ccc* field describes the condition; the *y* field describes whether the jump is on the condition true or false. The *xxxxxxx* field describes the number of halfwords to be branched; the *s* field describes whether this number is added to, or subtracted from, the PC.

Memory-Destination Ops

The common opcodes with general source and destination are described by the bit pattern

0 *aaa rrr x a u vv sss y*

The *aaaa* field, which takes on values from 4 to 15, contains the opcode. The 6-bit descriptor

vv sss y

describes the source value. The destination memory address is described by the 6-bit descriptor

1*u rrr x*

The opcodes which exist in this form include **add, subtract, and, or, exclusive or, move, compare, left** and **right shift, load under mask**, and **load address.**

Register-Destination Opcodes

Opcodes which may have their result in a register are represented by the form

1 *bbb rrr bbb vv sss y*

where *bbbbbb* is the opcode, the 6-bit descriptor

vv sss y

describes the source, and *rrr* describes the destination register. The ops included in this form include the ones with memory destinations, together with **signed** and **unsigned multiply, quotient,** and **remainder, call, save, return, simulate interrupt**, and **return from interrupt.**

Appendix II: A Possible 3-Address Machine

Some machines, noticeably the VAX, have the possibility of doing 3-address operations as well as 2-address operations. Since such machines have some advantages in code generation, it is natural to ask whether they are more efficient in the compilation of C code. This Appendix examines the effect of adding 3-address instructions to the basic design described in the main body of the text. To summarize the conclusions, 3-address instructions do not seem to save a dramatic amount of space, although they do save a little. Adding them does seem to complicate the machine considerably. The results of this section assume that the optimizations discussed above (for example, the Wheeler Device) have been carried out.

To completely evaluate a 3-address design would require that a compiler be built and numbers taken. However, it is possible to get a first-order estimate of the effect by collecting additional statistics from the output of the 2-address machine described above.

It is difficult to specify a useful number of 3-address opcodes in only 16 bits. The most natural way to deal with this restriction is to go to a byte-syllable form of organization, similar to the VAX machine. Thus we shall assume that the basic instruction on the 3-address machine will be a byte, and then an additional byte will be used to specify each argument. Some other assumptions are:

1. The registers and stack organization will be the same as the 2-address machine. Note that this implies the use of the Wheeler device.
2. The optimizations which are useful on the 2-address machine will also be applied on the 3-address machine. Moreover, there is the potential for address modes which allow a byte offset from a register, and byte immediate constants. It is assumed that the argument descriptors can describe a register or a 4 bit immediate constant within the byte, or may trigger the reading of one, two, or four more bytes of immediate or offset data. Branches are assumed to be optimized in a way similar to the 2-address machine.
3. All binary opcodes are assumed to have both 2- and 3-address forms.
4. The call operator is assumed to be followed by a series of operand syllables which describe the arguments; these are automatically moved onto the stack. Two forms of call are examined; in the traditional form, arguments are pushed onto the stack, decrementing the stack pointer for each argument. This requires an adjustment after the call: if a trick used on the PDP-11 is used, this adjustment need only be done if two or more arguments were pushed. In the alternative form of the call, the arguments are moved directly onto the first few integers on the stack. This avoids all need for adjustment of the SP after the call.

As a consequence of these assumptions, we see that the 3-address machine will win in space used whenever a 3-address instruction is to be used: a 3-address form of

a = b **op** c

is two bytes smaller than the "load, op, store" form that would be needed on a 2-address machine. In the case where an expression is computed, but not stored, the 3-address machine can compute it directly into the register in which it is required, saving a load. The space breaks even here. On real two-address operations, such as move and compare, the three-address specification loses a byte. When there is an immediate or offset which fits into a byte, but not into 4 bits, the 3-address machine saves a byte. Finally, with the traditional call strategy, the 3-address machine saves a byte with one-argument calls, loses a byte with two-argument calls (because of the stack adjustment), and saves $k-3$ bytes with k-argument calls, k three or more. With the alternative form of call, k bytes are saved for each call having k arguments.

The measurement tool which was used for 2-address measurement was adapted to collect some statistics which would suggest the magnitude of the savings/losses caused by these effects. The change in the call operator would lead to 1.7% saving if done traditionally, and a 5.5% saving if the alternative form were done. Byte immediates would account for a 1.3% saving, and byte offsets would yield 3.8%. Balancing this, an attempt was made to see what fraction of the 2-address operations generated were really capable of being turned into 3-address codes. This number is the most

suspect of the lot, since changes in compiler strategy might increase it; on the other hand, the number as counted is rather too high, taking count of some operations on register variables which could not, in general, be replaced by 3-address forms.

In the sample, only 2.2% of all the "real" operations (not calls, branches, saves, returns, or moves) could be converted into 3-address forms. However, many operations could save a load or a store by using 3-address opcodes; 4.8 % of the opcodes are moves which could be eliminated in this way. The 3-address effect is small, however; it only affects about 7% of all the opcodes. Over 17% of all opcodes were moves which could not be eliminated by using a 3-address form or the call operator; these require more space on a 3-address machine than on the above design. The result was a total penalty of 11 % for the moves and ops.

One other possibility which is attractive on a byte-oriented 3-address machine is to combine the test and branch operators into a single operator. While this gains little or nothing on a 2-address machine, for the sample this optimization would save 3.1%. A similar optimization is to have a unary form of test, which tests against 0. Since 4.2% of all opcodes generated are tests against 0, this optimization saves 1.3%. In summary, assuming all of the above optimizations, including the alternate form of call, were applied to the 3-address machine, the code generated by the 3-address machine would be about 4% smaller than for the 2-address machine.

Note that, with a test/branch combined instruction, most of the benefit of condition codes has been eliminated. If condition codes were eliminated entirely, the resulting program would have to grow about 3.5% in space in order to achieve the same result; considering the simplifications that are likely to result in the machine, this may be worth doing. On the other hand, it is likely that the time performance of the machine would be more affected by this than the space performance. Moreover, condition codes are occasionally useful for such applications as FORTRAN arithmetic IF's, and extended-precision arithmetic.

It must be stressed that these results depend on the 2-address optimizations, including the Wheeler device and the easy access to elements on the stack, as well as the alternate form of call. Actually, there is not much of a case to be made for or against the 3-address machine on the basis of these numbers. One could argue that a compiler tailored to the machine could lower the number of moves, and thus the space penalty. One could also argue that the 3-address machine being specified is considerably more complicated, in terms of addressing modes and extra instructions, and the added complexity seems to have little benefit in performance. In the absence of better data, however, deciding between 2-address and 3-address machines should be done on factors other than opcode efficiency.