Language directed computer design

by W. M. McKEEMAN
Stanford University
Stanford, California

An imaginary absurdity

It is an accident that digital computers are organized like desk calculators—with somewhat worse luck we might have taken the Turing machine as our model. And someone would have been unenlightened enough to prove that, under certain (actually untrue) assumptions, it made no difference. All general purpose machines can compute the same functions, given sufficient time.

Now, if one of the users of an automatic digital Turing machine were to suggest revolutionary additions such as a large random access memory and a special command to add the contents of two memory cells, we could expect to find him attacked on various counts. A design engineer would complain that the additions were *ad hoc* and destroyed the essential simplicity of the Turing machine. Besides they would make the machine ten times more expensive to manufacture. Another user would wistfully agree that the additions were clever and nice but he couldn't afford the expense of reprogramming his entire library of Turing machine programs; a working group would publish a list of accepted standards that were in jeopardy. And finally it would be pointed out that next year's Turing machine would be twice as fast.

The disgruntled user would, of course, take refuge in higher languages. He would use a compiler that would, when it saw the symbol '+', generate the necessary 175 Turing machine instructions required to add two adjacent bit patterns on his tape. He might, when desperation clouded his judgment, attend user's group meetings to make demands for extensions to the compiler that the manufacturer could not reasonably implement on a Turing machine. The following year the user would switch manufacturers and the manufacturer would fire ten programmers for assumed incompetency.

As time went on, the user might console himself with progress: the discovery of a new 167 instruction add routine; hyper-Turing tapes where the bits are recorded in frames 9 bits wide; the appearance of a pipeline micro-parallel Turing machine which, under special circumstances, could execute 27 simultaneous Turing machine operations; and finally, the Ultimate—two time-shared Turing machines working on the same tape.

The solution of some problems would still be beyond reach and the federal government would allocate funds for a really parallel effort—100 Turing machines arranged in a 10-by-10 array sharing 20 tapes on a grid, 10 across the rows and 10 down the columns. Since the machine could be shown to be potentially 100 times as fast for some problems, the best programmers in the world would stand in line to use it, thereby insuring its success by contemporary standards.

The intuitive feeling of our user (that the Turing machine model is not very good) is probably correct, but he must be quite careful in his attempts to demonstrate the feeling as a fact. Given the same level of technology, it is not obvious that the Turing machine does computation less economically. If, for instance, the active parts of the tapes are kept in high speed memory to avoid physical movement and the detectable possibilities for parallelism are fully exploited, automatic digital Turing machines might appear competitive with contemporary computers for a large class of problems. Nor can he claim that his programs are hard to write since the compiler takes care of those difficulties. Furthermore, small evolutionary additions to the Turing machine structure probably won't help much if we take into account all the costs of adding them.

The most obvious point of attack is that Turing machine code is highly redundant—it takes a great many more bits of code to represent a program than the information content demands. Thus the execution of large programs will be affected by the necessity of going to secondary store (or some other uneconomic means) for the instructions. Now compilers and

operating systems are large and frequently used, so they may be used as examples—everybody will agree that it is worthwhile to make them smaller and easier to write.

If the design engineers began to seek more efficient encodings for commonly used sequences of instructions (for instance, direct addressing instead of a sequence of tape moving commands), progress toward the modern computer would have begun.

Our challenge

Today, on contemporary machines, we observe that compilers and operating systems are getting less reliable, consuming more memory, taking more time and systems programmers to develop, use a larger percentage of machine execution time and produce substantially worse results. Keeping our Turing machine analogy in mind, it is interesting to examine the reasons for the trouble.

Since compilers and operating systems are just large programs, we are inclined to simplify the writing job by recourse to a systematic approach in a higher level implementation language. The fact is that most (but *not* all)* such attempts have failed because the resulting programs took too much memory. Since hand coded efforts have succeeded, it is fair to say that the compilers generally were not adequate for the job. Hand coding achieves compactness as a result of a multitude of careful decisions by programmers based on global knowledge of the functioning of the program. When those decisions are in error, we lose reliability. When they depend upon the functioning of some other code, we make program maintenance hazardous. In any case, it takes time to make the decisions and we see the effect in the increased cost of producing the programs. Even for the hand coded programs, the limiting property is size. A great deal of extra effort is spent in making large codes fragmentable, so that execution can proceed in stages with access to secondary store in between.

But it is absurd to expect carefully engineered, very fast, automatic desk calculators to be very good for implementing operating systems or compilers. We are forced to manufacture the operations we want out of sequences of hardware operations with the obvious result that the programs become large. When engineers begin to seek more efficient encodings for commonly used sequences of instructions, progress toward the modern computer may begin.

*Among the successful efforts we include all Burroughs B5500 compilers and operating systems, which are written in variants of Algol-60.

Our objective is to have the machines do our bidding reliably and with the minimum of effort on our part in telling them how. The most obvious, and most obviously profitable step is to make the machines reasonably amenable hosts for the operating systems, compilers and popular languages presently in use.

The palliative

The single most efficacious action that the computer manufacturers can take is to require that the key design engineers have personally implemented at least a production compiler and operating system for the previous machine. By "personally" we mean program design, coding, bit pushing and debugging clear through to a marketable programming system. Nothing less will do. As usual, a little knowledge is dangerous so it would even be valuable to have had him do some application programs. It is very discouraging to hear an engineer "prove" the correctness of a design by analyzing its performance in irrelevant circumstances. It would be somewhat worse to have a machine with some nice operation which cannot be used because it fails in a common but unsuspected circumstance. Careful design will see a total system simulation and completed operating programs before a single circuit is integrated.

The effects of this design approach should be immediate and salutary. It is rather easy to predict some of the results, direction and speed of the resulting designs.

Multiple arithmetic formats

First to go would be the multiplicity of arithmetic formats and the attendant complexities of conversions for every pair of types. We would also demand a consistent set of arithmetic operations that require identical register setups and leave their results in the same place. For example, a divide operation that leaves both remainder and quotient is occasionally convenient, but if that is the only divide that is available, it complicates our register allocation scheme during translation of arithmetic expressions. Only after an engineer has solved this problem in *his* compiler will he be able to say whether few types and consistent operations are worth an extra burden on the hardware.

A measure of the impact of this multiplicity of formats is seen by comparing the actual machine code required for a comparable FORTRAN compiler written on two different stack oriented computers. The stack oriented machine that provided the "programming flexibility" of many arithmetic and register

formats accessible to the programmer required *three* times as much code to implement this part of its compiler as did the machine with the clean, simple and consistent format accessible only through its almost pure stack structure.

A second measure of the programming consequences resulting from a machine design that utilizes many register sizes and allows many arithmetic formats within these registers is a comparison of the percentage of the total compiler required to manage and utilize this "flexibility." Consider a FORTRAN compiler, this time evaluated for a machine that has 4 different register lengths and 3 different arithmetic modes possible in each of these registers. The management, assignment, and manipulation of the 144 combinations of register conditions resulting from this "programming flexibility" required thirty percent of the entire compiler! The complexity required to handle this kind of problem in typical computers is a significant source of errors in code produced by the compiler.

As an aid to the hardware designers who are attempting to design their hardware to optimize software implementation, there are 4 major aspects of arithmetic and register symmetry that are needed:

(1) Different types or sizes of registers must be avoided.

(2) Multiple copies of the same register should not have to be addressed explicity.

(3) Multiple different uses of one kind of register should be avoided. For example, a typical base register oriented machine uses any one of a large set of registers for integer arithmetic, or for base addressing, or for subscript indexing, or for branch addressing. The management of these functions and their allocation among the many possible register locations is a big programming task, a big program always in memory, and it can be a very significant inefficiency at run time.

(4) If different arithmetic formats are used, the hardware should be capable of detecting and manipulating these formats so that the programmer and program need not take into account the different formats. "Typed Data" is an example of this consideration. Where more than one arithmetic format is desirable, we may find a few bits in each word which can be used to distinguish the type of the data. A single arithmetic operation could then be used to trigger not only the correct arithmetic algorithm but also automatic type conversions if necessary. Type bits can also be used to distinguish program from data as well as for many other operations where we want the hardware to be sensitive to the form of the data. The same mechanism can be used to handle access to subscripted variables, indirect addressing, parameters, and read and write protected memory.

Structured addressing

Next to disappear from commercial computers would be the single address instruction. A single address instruction that can use any word in memory is simply a license to commit mischief. In implementing a highly structured language such as PL/1 or ALGOL, we are less interested in addressing all of memory than we are in conveniently addressing all, and just, those variables that are presently active. The use of a stack and a set of register displays is well documented in the literature *as a compiling technique* but rarely found in hardware. Similar comments apply to subscripting and bounds checking, program segmentation, arithmetic temporary stores and subroutine linkage. In each case the machine designer can simplify the compilers and *speed the execution* of programs by confidently establishing some addressing conventions and dedicating some hardware to them.

Specifically for the benefit of our hardware designers, it should be noted that there is one remarkably simple addressing structure common to three of the major programming languages in use today. PL/1, FORTRAN, and ALGOL each use a pair of numbers to represent an address. This number pair has as its first digits the lexical (or nesting) level of the occurrence of the declaration of the name; and as its second digits the actual occurrence of the name in that level in the program.

For example, a typical program might be:

```
begin real A;
    begin real B,C;
    end;
    begin real D;
    end;
end.
```

Corresponding to the names above, the address pairs would be:

Name	*Address Pair*
A	1,1
B	2,1
C	2,2
D	2,1

B and D have the same address pair, and one can tell which name is referred by where one is in the program.

This language characteristic should lead the hardware people to reflect the fundamental elegance of modern languages into the hardware. What we need is a hardware system to compute the real hardware address at run time. For example, the lexical level could

point to a register, and the occurrence level could be an increment to that register.

The significance of this approach is that one is not likely to compile bad object code using this kind of structured addressing. As a result, a programmer addresses his own variables and only those that are immediately accessible. He not only cannot address other peoples' programs but he cannot inadvertently over-write even his own programs.

A second addressing structure which has a major impact on the size and thruput of software systems is addressing by pages vs. addressing by program segment. The basic question underlying this aspect of the discussion is what is the natural program segment size? Analyses of scientific programs we have done shows a peak in number of segments used at 60 words per segment. However, this is an ill defined peak lying between 20 and 100 words per segment, and even at that less than half of the total number of segments used lies in this range. The typical system designer has chosen to impose an artificial quantum of program to address by selecting a page size. Several problems in the programming systems immediately arise and should be noted by the hardware people.

First, in selecting this unnatural cleavage plane in a program, the hardware forces the system programmer into considerable overhead manipulation to handle and bridge the arbitrary cleavage that occurs unpredictably in all programs at run time. If the programmer ignores these artificial boundaries, the overhead goes up catastrophically.

Second, if a large page size is chosen compared to the actual segment size that is located in each page at run time, a considerable part of total memory is wasted. It is an accident if the remainder of the contents of that page happen to be needed unless the programmer has carefully arranged it.

Third, if a page size is chosen that is smaller than the actual segment to be run, there is lost time as the executive program handles the larger number of page faults.

The answer is to use a varying page size equal to the actual size of the natural program segment to be run. This approach is addressing by segment.

A comparative measure of the relative value of these two approaches has been estimated based on programs written first for a modern page addressed machine and second for a contemporary segment addressed machine. This comparison shows that for equal system overhead time measured in percent, that the machine using segment addressing uses approximately ⅓ the memory required by the page addressed machine. This represents a gain of factor of three in terms of main memory utilization!

This difference is of major significance in any commercial system, of course; but in practical terms this is a graphic example of the impact of matching the addressing structure used in the hardware to that used in the programming language.

Secondary benefits of matching these addressing structures occur in the areas of program bounds (or limit registers), in handling arrays, and in handling subroutines. As long as a program stays within its own limits in a page addressed machine, this program can destroy itself or its own subroutines. By contrast, in a segment addressed machine, program cannot do this to itself.

To summarize this section on structured addressing, these are but a few of the many ways for the hardware designer to use the language structures which have been invented to simplify writing and execution of programs.

Algorithmic structures

Some programs, the compilers and monitors in particular, may profit from special attention. A compiler-tested engineer might want a hardware scanner for his next effort; a monitor-tested engineer would likely ask for hardware queues and variable length segmentation structures. It may even be feasible for some programs to disappear completely into the hardware.

An example of the kind of hardware needed to substantially improve compiler thruput would be a hardware scanner. A scanner in programmer's nomenclature is a means for recognizing the character boundaries of the next message to be fed the compiler. This next message is called a token, and a token is an identifier, a reserved word, a special character, a number, or a character string. The means that programmers have used to find the next message, or token, is a programmed routine to scan the sequence of characters waiting to be compiled to identify and mark these boundaries. The characters are passed on as a group to the main part of the compiler. Historically, this scanner routine is the slowest part of compilers. Hardware assistance in this area, again based on the way the programming language is constructed, could provide significant reductions to the programming task.

The idea we are describing here is for the hardware designers to study the algorithmic structures of the important programs that will be written for or run on their systems. There are undoubtedly many system and hardware concepts which can be invented to take advantage of their software algorithmic structures.

As an aid to the hardware designers, some references to the relevant literature are:

Randell & Russell, *ALGOL 60 Implementation*
Academic Press, 1964
Marvin Minsky, *Computation*
Prentice Hall, 1967, page 117
Wirth & Weber, *Euler: A Generalization of ALGOL 60*
Comm–ACM, January, 1966

None of these ideas are new. Nor will they be particularly startling to programmers. The amazing thing is the lethargic pace of the industry in building machines that are directly designed to do the tasks they will be assigned. The measure of success will be in total system performance. It is incredible how few computer customers ever actually measure machine performance for the kind of job mix they normally run before they purchase a new machine. Granted that it is a difficult job; granted that we frequently buy hardware that exists only on paper; granted that the decision is frequently made on other grounds; nevertheless customers and manufacturers must begin to measure actual *performance* if we are to make any kind of progress in system design.

A remedy

The obvious reward of the palliative will be increased performance; the real reward will be the widespread realization that we are not at the end of the line and there are more constructive things to do than argue morbidly about standardization of languages and machines we don't fully understand.

Having surmounted the political problem, we can return to a direct attack on the central problem: How to make the machines do our bidding.

We are our own best model for computer organization. It is our work that we want the machine to do and we have some idea of how we go about it. It is apparent that we can greatly improve the performance of our systems by allowing and controlling unboundedly parallel operation. This system goal implies the invention of a class of artificial languages within which that control will be conveniently expressed. We will be better off if we take natural language concepts as our model than those of conventional programming languages; no matter how machine independent we wished our programming languages to be, they all have been overwhelmingly sequential, arithmetic and random access memory oriented.

The obvious attach for programmers and hardware people together is to devise language that reflects what we want to do and how we do it (for instance, in parallel) and machine structures effective in handling that language. Let us call this method "language directed computer design."