A perspective on the 801/Reduced Instruction Set Computer

by M. E. Hopkins

From the earliest days of computers until the early 1970s, the trend in computer architecture was toward increasing complexity. This complexity revealed itself through the introduction of new instructions that matched the application areas. Microcode was an implementation technique that greatly facilitated this trend; thus, most computers were implemented using microcode. In 1975, work began at the Thomas J. Watson Research Center on an experimental minicomputer. This project, termed the 801 project, questioned the trend toward complexity in computer architecture. It was observed that most of the complex instructions were seldom used. Thus, a computer could be designed with only simple instructions without drastically increasing the path length or number of instructions required to implement an application. This made it possible to implement a machine without resorting to microcode, which improved performance. This paper described the background and evolution of these ideas in the context of the 801 experimental minicomputer project.

Computers are unlike other tools in that they are truly general-purpose instruments. The interface seen by an airline ticket agent is that of a machine that makes airline reservations. A secretary sees a text-editing machine. To the applications programmer, who uses a high-level language like FORTRAN or COBOL, the computer is a FORTRAN or COBOL machine. However, the assembly language programmer and compiler writer see the underlying machine architecture, an IBM System/370, a DEC VAX, or some other instruction set.

A *computer architecture* is an abstract description of a machine; interest in computer architectures transcends any particular machine. Thus, there can be many models of System/370 or VAX, all of which can execute the same program. Because performance is so important, computer architecture is always done with a particular set of implementation techniques in mind. Register files, buses, and adders have similar properties in a wide variety of realizations, and implementation techniques such as parallel operations have wide applicability. The details of the various realizations may also influence the architect's decisions. For example, the architect may have to specify what happens in the case of a parity error, an event that does not occur on a purely abstract machine. On the other side of the architectural interface, the architect must cater to the needs of those who construct programs. This paper examines some issues in computer architecture in the light of an experimental processor known as the 801/RISC (reduced instruction set computer).

Background. Because they lacked clear criteria, computer architects between 1946 and 1975 tended to specify ever more complex systems, as hardware capability increased and the tools used for design became more powerful. The computers acquired more instructions, more addressing modes, more data types, and more special features. The experience of IBM is not very different from that of the rest of the industry in this respect. The IBM Type 701 was introduced early in the 1950s. It was truly minimal, because any complexity would have reduced reliability. Floating-point arithmetic was done with subroutines. By the time the IBM Type 704 was intro-

Table 1 Ten most-used instructions in a typical instruction mix

Operation Code	Instruction Name	Percentage of Total Executions
BC	Branch	20.16
L	Load word	15.49
TM	Test under mask	6.06
ST	Store word	5.88
LR	Load register to register	4.70
LA	Load effective address	4.04
LTR	Load and test register	3.78
BCR	Branch on register	2.69
MVC	Move characters	2.10
LH	Load half word	1.88

duced in the mid-1950s, it had instructions that performed floating-point arithmetic. Providing floating-point arithmetic by an instruction as opposed to a subroutine offered no new function. However, performance improved in two ways: (1) The new engine had circuitry to perform efficiently floating-point functions such as normalization that required many basic cycles on a 701. (2) By performing the high-level operation entirely in the CPU, the transfer of instructions between memory and the CPU was reduced.

This notion of adding new instructions tailored to the expected application became a guiding principle of computer architecture. By the early 1960s, IBM was producing families of computers oriented to their expected application area. The IBM Type 7090 was the scientific and engineering successor to the 704. It had fixed-word binary and floating-point instructions, whereas the 7080 had variable-length decimal arithmetic and edit instructions suited to commercial work. By this time, much programming was being done in high-level languages like FORTRAN and COBOL. Both machines had FORTRAN and COBOL compilers. Unfortunately, there were no systematic methods of comparing architectures. However, anecdotal evidence suggested that the 7090 actually executed commercial applications faster than the 7080, even though both machines used the same circuit technology. Nobody examined this situation, perhaps because software and the portability of customer applications were deemed to be problems of higher priority that urgently required solutions.

The solution was System/360, a machine that coalesced all applications. The technology to implement a machine with many diverse instructions was provided by very fast read-only storage that permitted the economical implementation of microcode.[1] This ushered in a new era of machines with a great diversity of instructions. In addition, fundamental improvements were made in the treatment of the memory hierarchy. Virtual memory was introduced on the System/360 Model 67 and caches on the Model 85. The general direction of both architecture and implementation was toward greater complexity.

In 1975 the 801 project was started at the IBM Thomas J. Watson Research Center. (The project was so named because the Research Center is Building Number 801.) Our intention was to re-examine the trend toward complexity. The 801 was to be a hardware and software system. Almost all programming would be done in a high-level language. The goal was to discover how to deliver the most computing power at the lowest cost in an environment geared to programmer productivity. A prototype version of the 801 was built and system software was written. This included a compiler known as the PL.8,[2-5] which became a tool and a means of studying the effectiveness of the 801 architecture. Projects with similar goals have also been pursued at the University of California at Berkeley and at Stanford University.[6] This architectural point of view has become known as RISC, from Paterson's Berkeley Reduced Instruction Set Computer (RISC) computer.[7,8]

The term RISC nicely captures the spirit of research into the premise that less is better. The various aspects of architecture addressed in the 801 project are discussed in References 9 and 10. However, our goal was not to have the fewest possible instructions, but to simplify the machine's data flow to make the basic instructions run faster. Also, we had observed that mechanisms such as virtual memory and caches had been separately introduced on System/370, and we thought that this project might discover whether a more consistent and efficient implementation might be possible if a fresh start were taken. Adequacy to implement a broad spectrum of high-level languages efficiently was also a concern. This paper examines the reasoning that led to the experimental 801 architecture.[11] Many of the lessons learned during the project have been incorporated into Romp, the CPU for the IBM Personal Computer RT.[12]

System/360 instruction traces

An illuminating insight into computer architecture was provided by the first instruction traces on System/360. The motivation was to perform an experiment to determine an optimal geometry for the cache on what would become the Model 85. However, the determination of frequency of instruction usage was a natural by-product. It is easy to see that most instructions on a machine with about 200 instructions will be executed very infrequently. It came as a surprise, however, that load, store, branch, and a few simple register operations almost completely dominated the mix of instructions. The ten instructions that ranked highest in percentage of total instructions executed are given in Table 1.

Together these ten instructions capture two thirds of all instructions executed, a discovery that raises several questions. If storage-to-register add and add logical are not among the top ten, is it worthwhile to have instructions that combine the two relatively basic functions of fetching a value from storage and then performing an arithmetic or logical operation? System/370 has a comprehensive set of instructions that fetch one operand from storage and the other from a register and put the result of the operation back into the register. Together these instructions, which include add, subtract, compare, AND, OR, and exclusive OR, constitute less than 6.5 percent of all executions. Also consider other complicated instructions. Load multiple and store multiple together account for 2.4 percent of all executions. Integer multiply is 0.115 percent, and divide is 0.111 percent of all executions. The loop-closing operations BXLE and BXH are 0.066 percent. With the exception of the instructions pack and convert to decimal (0.008 percent), the decimal and edit instructions are not represented on this trace.

We then considered how representative of the range of actual customer environments this trace might have been. If small snapshots are taken, differences based on the application and compiler or on the human coder who wrote the program can certainly be seen. There are scientific traces for which you can almost see the code for the inner loop of matrix multiply as produced by a particular FORTRAN compiler. In general, scientific program traces look a little different from others. In extreme cases, they may have 10 percent or more floating-point instructions. There may be other differences, such as significantly fewer of the instructions associated with linkage—BAL (Branch And Link), for example. However, it is surprising how often scientific programs spend most of their time doing integer calculations. In many cases, a large proportion of their execution seems to be spent in the input/output and formatting routines. Knuth has an interesting discussion of this phenomenon.[13] In the late 1960s, short object code traces of code produced by the COBOL compiler sometimes showed a measurable number of decimal instructions. In recent years, this has become less true; a number of factors account for this. Commercial applications have become more diverse. Very little decimal processing is required in an inventory control application. More of the CPU cycles are now concerned with data base, network, communication, and screen formatting, all of which is systems-type code and is often performed by some subsystem. As the control program takes on more functions, it is inevitable that there will be less decimal computation. Thus, there is an overall tendency for traces of program execution to become like systems code. With the exception of floating-point, it is very difficult to identify an instruction that is potentially important, but only in a limited set of environments. The existence of trace tapes tends to inhibit the introduction of new instructions in existing architectures. For example, will any newly proposed instruction be executed more frequently than the existing Translate and Test instruction (0.001 percent)? It is also a useful exercise for those proposing a new architecture to construct hypothetical typical mixes.

Table 2 **Example trace results for storage-to-storage logical operations on System/370**

Operation	Frequency of Use
NC AND	0.050
OC OR	0.058
XC exclusive OR	0.555

We also considered execution time as opposed to frequency. Obviously, the more complex the instructions, the greater the execution time. Time taken is a function of a particular implementation, so we cannot state general rules. However, one observation has been that programmers and compilers sometimes choose a single operation that is relatively slow in preference to a sequence of faster instructions. For example, the average length of an operand of the Move Characters (MVC) instruction that performs storage-to-storage move on System/370 is less than seven bytes. Thus there must be a number of one-, two-, and four-byte moves. On virtually every System/370 implementation the Load-Store sequence is considerably faster than MVC for short align moves.

The trace tapes also provide information to those designing machines. For example, the frequency of

Figure 1 General features of the System/360 design

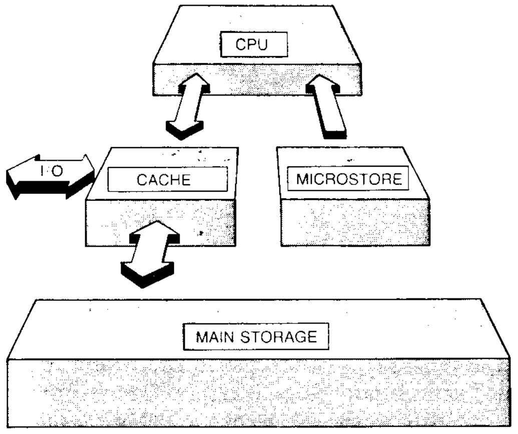
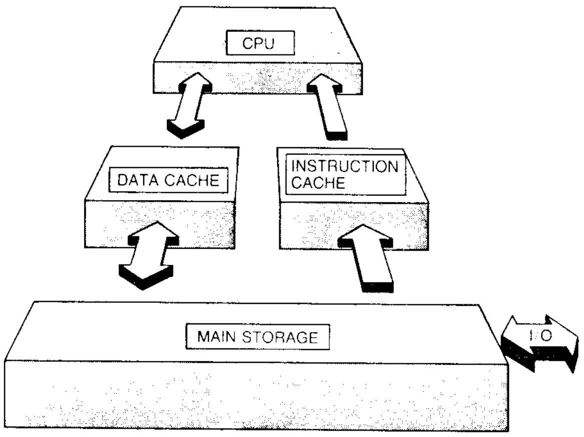


Figure 2 General features of the experimental 801 Reduced Instruction Set Computer



storage-to-storage logical operations in one System/370 trace is given in Table 2.

Why is the XC instruction used ten times more frequently than NC or OC? The explanation is that an exclusive OR of a variable to itself provides a means of setting storage to zero. Machine designers can and do capitalize on this fact. Thus both operands that coincide run more than twice as fast as operands that are disjoint. For those developing a new architecture, the lesson may be to provide a fast way to zero storage rather than providing storage-to-storage logical operations.

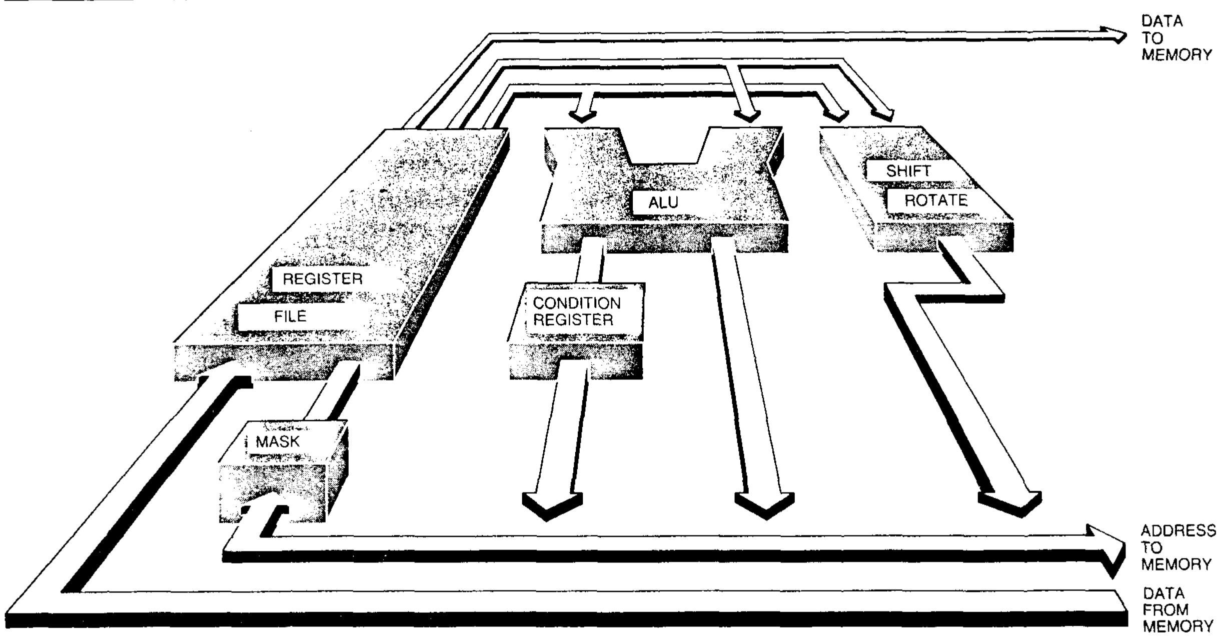
Although many insights can be gained from trace tapes, the important fact to the 801 project members was that, in practice, the simple instructions are the most frequently used and in that sense are the most important.

Considerations leading to the 801 experiment

In 1975, when the 801 experiment project began, most System/370 machines were implemented with a cache and microstore, as shown in Figure 1. The cache had been introduced to provide faster access to main storage, that is, to bring the speed of memory more in line with that of the CPU. The CPU fetched instructions from the cache and interpreted them on the basis of a microprogram in the microstore. An average System/370 instruction might take 20 or 30 microinstructions. Neither the cache nor the microstore is part of System/370 architecture, but perhaps they provide an architectural opportunity. The cache and microstore could be, and often are, implemented in the same technology. Consider why the frequently executed first-level-interrupt handler (FLIH) and the task dispatcher are executed out of the cache, whereas decimal divide is executed out of microstore. One approach is to make key system facilities into high-level, complex machine instructions that are executed out of the microstore. This has the advantage of permitting the chosen functions to be implemented faster because the microinstructions can each be executed in a single cycle. Because the FLIH and dispatcher are implemented primarily with simple operations that have a direct counterpart in microcode, there is a potential for a 20-to-1 improvement in performance. Further gains may be realized by reducing traffic to main storage, because the microstore has a separate path to the CPU. Perhaps even more instructions might be designed to exploit this advantage. Such instructions reflect the design of the system software and the requirements of compiler writers. As more specialization is introduced, it is inevitable that the microstore will become larger. Eventually it may be necessary to make it writable and page it like the cache, using some least recently used (LRU) replacement scheme.

But why do we permit only the microcoders to access the microstore; can it be made available to all? If we do, there will be a new interface with many of the same properties of our current computer architecture. As more microcode is written, there will be a need for microcode compilers. Such questions as whether the microcode compilers have an option to do subscript range checking will have to be answered. The microcode instruction interface will have to be precisely defined to enhance portability to protect a

Figure 3 Idealized data flow in the experimental 801 Reduced Instruction Set Computer



large investment in microcode. Great care will have to be taken to ensure that it is safe, easy to use, and a good target for compilers. Because good performance is obtained by using microcode, the performance problems of many users will have to be in the hands of the microcoders. The 801 group chose a different approach to a solution. An architecture was specified that would not—as System/370 does—tacitly assume an implementation in microcode.

The assumed 801 experimental system design is shown in Figure 2. In the architecture, the only built-in instructions are those that can be implemented in a single cycle, with the simple sort of CPU that was used to interpret the System/370 instruction set. The microstore has been exchanged for an instruction cache. Figure 3 gives an idealized data flow for such a machine. Such an approach provides a pervasive performance improvement for all executions of the simple instructions that the trace tapes have shown to be most frequent. The overall system design of Figure 2 also provides advantages over the microcode implementation in Figure 1. By providing separate data and instruction caches, the bandwidth to memory is potentially doubled. One of the problems encountered with traditional cache architectures is that the I/O is run through the cache. Thus the I/O activity tends to fill up the cache. The experimental 801 architecture, by running I/O directly to memory, tends to increase the memory bandwidth even further. Notice that the chief justification for the cache and microstore in Figure 1 was to reduce memory traffic; the 801 solution is an improvement upon this.

There is a disadvantage with this split-cache approach. What happens if an instruction is fetched into the instruction cache and, while there, its main-storage reflection is copied into the data cache and modified? This is an example of the need for cache synchronization. Input-output has a similar problem. In practice, nobody modifies instructions; thus, the 801 approach is to abandon the practice. However, there are some programs that must construct programs. Loaders are the most conspicuous example. The 801 experimental architecture makes the notion of a cache explicit; those programs that construct programs must issue instructions to synchronize the caches. The vast majority of programs need never be concerned with cache synchronization, but its use is required in a few areas of the system for programs to function correctly. The 801 requires software synchronization. In the light of current software practice, this imposes a modest burden on a very few systems programs in return for a substantial improvement in the price/performance ratio.

Having specified the 801 architecture with the tacit assumption that an implementation will have a split cache, we do not believe that we have unduly restricted the freedom of implementers. Both a cache

A shifting of responsibility between hardware and software was recognized by the 801 experimenters.

and noncache implementation of an 801 must cope with the fact that storage operations and branches are inherently slower than register operations. We first describe how the 801 architecture ameliorates this and then consider why registers are faster. The basic idea with loads is to use the arithmetic and logical unit (ALU) to compute the *effective address* in one cycle. This effective address is then sent out to the memory subsystem. In order to avoid the cost of examining page tables in memory, most memory subsystems maintain a hardware look-aside that maps recently used virtual addresses to real memory. This is called a translation look-aside buffer (TLB). As TLB miss ratios of under two percent are common, we concentrated on the 98 percent of references that hit the TLB. The memory system would be given the real address by the TLB and fetches would be given the data from memory. After dispatching the address, the CPU locks the target register of the load and proceeds to execute the next instruction(s), unless they use the target register, in which case the CPU waits until the load is complete. For this strategy to be effective, compilers and assembly-language coders must arrange instructions to ensure a maximum of overlap. Notice that even instructions of very modest complexity would inhibit this process. Suppose it is desired to add X in storage to register RX, and add Y to register RY. Assuming that it takes two cycles to get to the storage data, which are presumed to be in the cache, and one cycle to do the operation, on System/370 we would have the following code:

```
A RX,X     2 cycles to fetch
           1 cycle to add
A RY,Y     2 cycles to fetch
           1 cycle to add
           ----------------
           6 cycles total
```

On the experimental 801, the code would be the following:

```
L RO,X        2 cycles to fetch
L R1,Y        1 cycle not overlapped
A RX,RX,R0    0 cycles
              (overlaps previous load)
A RY,RY,R1    1
              --------------------
              4 cycles total
```

Although the 801 code has more instructions, it runs faster. Of course, one can adopt similar strategies on System/370. The experimental 801 solution is simpler in hardware. It does require that the compiler writers make the effort to schedule instructions, as such local rearrangement of code is called. It is possible, at considerable cost in hardware (not microcode), to dynamically rearrange the components of the multifunction instructions during execution to achieve the 801-type overlap. The trouble with this kind of solution is that, even if one can afford the hardware, complexity has its own price, and the ultimate result of many such solutions is a slower basic cycle. If the cost of a more complex implementation is an increase to 11 levels of logic from 10 due to chip crossings or such considerations, then the machine is 10 percent slower overall and this was done for 6 percent storage to register computational operations. The requirement to schedule instructions is an example of a shifting of responsibility between hardware and software that was recognized by the 801 experimenters, even though it is never explicitly mentioned in the architecture.

It is now appropriate to examine why a cache is inherently slower than registers. Access to a cache requires two sets of translation. First, the effective virtual page must be converted to a real page by accessing the TLB. Then the memory subsystem must determine whether the cache line within the page is also within the cache. Such lookups are normally done by hashing the address and doing a comparison. (On some architectures the cache can be kept virtual, which permits the cache lookup to be done in parallel with the TLB lookup.) Further complications occur because misses and storage key violations are possible. In general, the construction of a memory subsystem is highly complex. A good summary of implementation techniques is provided in Reference 14. The complexity of accessing a cache can be contrasted with accessing a register, the name of which is in the instruction itself. In a properly architected machine, it is possible to begin fetching both register operands before the operation has been decoded. If a register operand is not needed, it can be discarded. The key architectural requirement is always to place the source register names in the same positions within the instruction format. The existence of instruction formats that permit great variability in the placement of register names is one of the major barriers to improving performance on some machines. In general, register access is so simple that it is feasible to fetch two operands out in a single cycle. In the idealized 801 in Figure 3, there are three paths out of the register file. This permits the CPU to do all its work for a store in one cycle, although the memory subsystem may consume more cycles. The quantity to be stored can be fetched as well as the base and index registers, which are added to compute the effective address. There are also two paths into the register file. The first is for normal results from the ALU. The second permits the result of a load to be returned to its target register in the same cycle as a register-to-register operation. With three outputs and two inputs, a register file can be viewed as having five times the bandwidth of a cache, and there is no need to go through a virtual-to-real translation and cache lookup.

Branches are also inherently slower than pure register operations. When instruction execution is sequential, it is relatively easy for the CPU to overlap the execution of the current instruction with the fetch of the next, but a taken branch interrupts this process. Some larger machines keep branch history tables to predict branches. However, the 801 provides an architectural alternative to assist the implementers. The basic idea is to define a companion operation for every branch instruction that executes the next (subject) instruction in parallel with the branch. These are called "execute branches" on the 801 and "delayed branches" on the Berkeley RISC. As the outcome of the branch cannot depend on the results of the subject instruction, the implementer can design hardware to overlap execution of the branch and the subject instruction.

Such hardware is not as complex as that required for branch overlap on high-performance machines, but it is far from simple. The chief complications result from cache and TLB misses on the branch itself, the subject instruction, and the branch target. It is interesting to do a quick analysis to determine the potential value of this type of branch. We assume two cases: A taken branch costs (a) two cycles and (b) five cycles. This bounds the problem for high-performance cache machines and simple machines without caches. We assume the same distribution of instructions in each case. Consider the case of the two-cycle cache first in Table 3, which assumes that three quarters of all branches can be converted to execute form. Of these, two thirds cover storage operations, and the rest cover register operations. As can be seen in Table 3, this set of assumptions provides a 10 percent performance improvement. The 18 percent saving obtained on the cacheless implementation shown in Table 4 is even more dramatic. The assumptions behind these improvements can be questioned. In particular, the average storage instruction may take less time because of overlap. One can also question the mix, or whether three quarters of the branches can be converted to execute form and whether two thirds of the subject instructions will be storage operations. However, for a wide range of reasonable assumptions, execute branch seems to be very cost-effective. It is common for those designing machines to invest great effort to obtain performance improvements of less than one percent, and here we see a potential gain of 10 to 18

Table 3 Effect of execute branches on one hundred average instructions using two cycles

Case with No Execute Branches

Instruction Type	Number of Instructions	Cycles per Instruction	Total Cycles
Branches	20	2	40
Storage	30	2	60
Register	50	1	50
Total			150

Case with Execute Branches

Instruction Type	Number of Instructions	Cycles per Instruction	Total Cycles
Branches	5	2	10
Execute branches	15	2	30
Storage	20	2	40
Storage (subject instructions)	10	0	0
Register	45	1	45
Register (subject instructions)	5	0	0
Total			125

Table 4 Effect of execute branches on one hundred average instructions

Case with No Execute Branches			
Instruction Type	Number of Instructions	Cycles per Instruction	Total Cycles
Branches	20	5	100
Storage	30	5	150
Register	50	1	50
Total			300

Case with Execute Branches			
Instruction Type	Number of Instructions	Cycles per Instruction	Total Cycles
Branches	5	5	25
Execute branches	15	5	75
Storage	20	5	100
Storage (subject instructions)	10	0	0
Register	45	1	45
Register (subject instructions)	5	0	0
Total			245

percent. There is, however, a software price to be paid. The compiler writers will have to produce execute branches. One way to do this is as follows. In the compiler phase that rearranges code to maximize overlap on loads, an attempt is made to move a suitable instruction next to each branch. Final assembly then flips the branch with the previous instruction if there is no interdependence. In practice great care may have to be taken. Consider the code that typically closes an iterative loop:

```
AI     R1,R1,1   Bump R1
CI     R1,100    Test if limit exceeded
BLE    LOOP      Branch on low or equal
```

Given such a sequence, it may not be possible to find a suitable subject instruction.

Under the right circumstances, the code can be reordered and the loop-closing instruction altered to test for inequality as follows:

```
CI     R1,100    Test if limit reached
AI     R1,R1,1   Bump R1
BNE    LOOP      Branch not equal
```

Final assembly can then flip the AI and the BNE to produce the following sequence:

```
CI     R1,100
BNEX   LOOP
AI     R1,R1,1
```

Notice that there must be a form of add that does not set the condition code for this to work. Such details are extremely important.

This transformation is not the end. Consider the following standard loop in C that does a storage-to-storage move:

while (*t++=*s++);

For those who do not read C, this statement moves a string of characters located by the pointer s to the storage located by the pointer t, one byte at a time, terminating when a zero byte is moved. In the process, the pointers t and s are incremented on each iteration of the loop. The following is some code produced by the PL.8 compiler in Yorktown, with somewhat idealized cycle counts on the right:

```
LOOP  LCS    R0,0(RS)    5
      INC    RS,1        1
      STCS   R0,0(RT)    5
      CIS    R0,0        1
      BNZX   LOOP        5
      INC    RT,1        0
                         —
                         17 total cycles
```

On the surface, it may seem that it is impossible to improve this code. However, the store character instruction can be made the subject of the BNZX, as follows:

```
LOOP  LCS    R0,0(RS)    5
      INC    RS,1        1
      INC    RT,1        1
      CIS    R0,0        1
      BNZX   LOOP        5
      STC    -0,-1(RT)   0
                         —
                         13 total cycles
```

The trick is to have the compiler move instructions that modify the base address by a constant, altering the displacement field to reflect the add. This transformation provides a saving of almost 25 percent on the most common idiom in C, and it is also used in many similar situations.

The subject of execute branches provides a good example of the following range of considerations that are pertinent to architectural decisions:

- Instruction frequencies
- Hardware implementation techniques in a variety of situations

- The way in which instructions are actually used, including linguistic idioms
- Compiler design
- Details such as which instructions should set the condition code

Having these kinds of data, the architect must evaluate whether a proposal such as branch and execute is worth the inevitable hardware and software implementation problems, given the expected usage patterns. Execute branches are a graphic example of the partnership among hardware implementation, architecture, and compiler design that took place in the experimental 801 project.

Except for loads, stores, and branches, all the instructions can be implemented to execute in one cycle.

High-level function

The 801 experimental architecture provides a set of primitive instructions. Except for loads, stores, and branches, all the instructions can be implemented to execute in one cycle on a simple implementation. Now consider the function performed by more complex operations, such as those that exist on System/370.

Integer multiplication provides an example of the variety of techniques used to implement high-level function on the 801. We first assessed the extent of the problem. Typical System/370 traces show that all forms of integer multiply constitute less than 0.15 percent of all executions. Depending on the model, it takes between five and thirty times as long to do a multiply compared with a register add on System/370. Considerable hardware is required to achieve the lower number. In practice, it is often achieved by doing integer multiply in a very expensive floating-point unit. The most economical way to implement multiply is to have a 2-by-32-bit microcode multiply step instruction. The System/370 multiply instruction is then implemented by executing 16 such operations in microcode.

The experimental 801 architecture provides a multiply step instruction in its instruction repertoire. The control program provides a multiply subroutine that is shared by all users. It consists of 16 consecutive multiply-step instructions. By software convention, the multiplier and multiplicand are passed, and the product is returned in registers. To the extent that multiply is frequently used, the code is in the cache; thus, the proportional time is roughly the same as that of the microcoded System/370 implementations, even if we include linkage costs. This should not be a surprise, because the same work is being done. An important difference is that the microstore and multiply-step instruction are only available to the microcoder, whereas on the 801 the instruction cache and multiply-step instruction are available to all. Suppose there is a problem that requires only a 12-bit multiplier. In this case, it is possible to write a 12-bit multiply subroutine, which remains in the cache to the extent that it is used. Such a routine can be further specialized to handle negative operands in special ways and to check ranges. In effect, the programmer can define his own variations on multiply, and it is difficult to see why they should perform very differently than a microcoded routine in the same technology. It is even possible to put multiply-step instructions in line, providing the effect of in-line microcode.

In practice, many multipliers are constants derived from subscript calculations. Most compilers convert multiplies by power-of-two shifts to the left. On the experimental 801-type machines, it is common practice to convert all constant multipliers to a series of shifts, adds, and subtracts. In practice, it is rare to require more than three or four instructions, so that it may be possible to exceed the performance of an expensive high-speed multiply unit for most cases.

In practice, both compilers and human coders try to reduce the number of costly instructions. One example is to have the compilers do strength reduction. This is a compiler optimization that replaces multiplies that result from subscript calculations with adds. Consider the following program fragment:

```
declare
   x(0:100)        character (50);
do i = 0 to 100;
  if x(i) = y      then
   leave;
end do i;
```

As each element of the array x takes fifty bytes, a reference to $x(i)$ requires multiplying i by fifty. The process of strength reduction introduces an auxiliary variable i', which is set to zero at the beginning of the loop and incremented by 50 on each iteration, eliminating the need for a multiply. Such optimizations apply to the experimental 801 as well as to more complex machines. Their tendency is to reduce the benefit that might be obtained by having complex operations.

Partly because of the lack of high-level instructions but also because the use of subroutines is so central to much of good programming practice, a great deal of attention has been given to providing efficient linkage on the experimental 801. There are three classes of branch and link instruction:

- Relative branching is used within a bound module. A 32-bit instruction can generally be made to accommodate 20 to 26 address bits. This establishes a maximum size for link-edited modules of 1 to 64 megabytes.
- An absolute branch is required to get to shared supervisor routines, such as multiply or storage-to-storage move. The address field of the instruction is of limited size, so such microcode routines have to be located in the first few megabytes of virtual memory.
- A branch and link can be done on the contents of a register in situations in which the invoked routine is computed at run time.

The first two forms of Branch And Link (BAL) require no load to link to a subroutine, a significant saving given that BAL constitutes between 1 and 2 percent of all instructions executed.

The number of available registers and how they are used are central to the efficiency of linkage. Because intermediate results are maintained in registers and can be retrieved rapidly, it is a good idea to have a fairly large number of registers. The number of bits required to name a register in an instruction is an architectural constraint, but there are usually also technological limitations on the number of registers. A system constraint is the necessity to save and restore registers when there is a process switch. The first experimental version of the 801 was built with 16 registers. Studies of code produced by the PL.8 compiler showed that over 50 percent of all the procedures in a large sample had some register *spill code*, which comprises the load and store instructions that would not be required if there were more registers. Because the technology made a 32-register machine possible, we decided to define the next experimental 801 with 32 registers. Because more bits were required to name the registers, this necessitated going to a machine that had only 32-bit instructions. The 801 prototype had 16- and 32-bit instructions. Uniform instructions have advantages beyond permitting more registers. It is possible to define all register operations so that there is no need to destroy one of the operands. Having a uniform instruction size also simplifies, and thus speeds, the process of instruction fetch. For example, there can never be two cache misses when fetching an instruction.

Having a uniform instruction size also simplifies, and thus speeds, the process of instruction fetch.

Reference 12 gives a detailed description of linkage and register conventions on a 16-register computer chip known as Romp. We give a summary for a 32-register machine. Up to six parameters are passed in registers; the rest are passed in storage. Studies have shown that most procedures have very few parameters, which means that a called routine can usually use the parameters directly in the registers, and the caller need only go to storage if the value passed has to be fetched. For languages like FORTRAN, which do calls by reference, parameters are pointers to the argument. For languages like C that have value parameters, the actual values are passed in registers. These six registers and perhaps a few more will be assumed to be destroyed over a call. One register is dedicated to pointing to the stack frame, and it may be necessary to dedicate another register to a process communication area or similar region of storage. Thus, there are about 20 registers that must be preserved over each procedure call, whereas about ten may be altered. A routine that alters only the first ten registers need not save any registers. Paradoxically, having more registers in the hardware can mean fewer saves. With this approach it is possible to have very simple linkage for simple subroutines.

Figure 4 C-language loop to do a storage-to-storage move

```
    while (*t++ = *s++);
    return;

Object code for Romp

5| 000000                        %6:
5| 000000 LCS    4003                      LCS     r0,$MEMORY+*s(r3)
5| 000002 INC    9131                      INC     r3,r3,1
5| 000004 INC    9121                      INC     r2,r2,1
5| 000006 CIS    9400                      CIS     cr,r0,0
5| 000008 BNBX   89AF FFFC                 BFX     cr,b26/eq,%6
5| 00000C STC    DE02 FFFF                 STC     r0,$MEMORY+*t-1(r2)
7| 000010 BNBR   E88F                      BFR     24,r15
```

Figure 4 shows the PL.8 code for a subroutine that does the C storage-to-storage move on the Romp processor. There is no prologue, and the epilogue is a branch register. More complex programs and the requirements of different languages necessitate more linkage instructions, but it is important that this cost be incremental. The 801 approach, unlike the introduction of powerful complex operations, permits specialization to what is required in a particular situation. There is no need to do extra work. Note that there is a tacit assumption here that most linkage code is to be constructed by compilers. Conventions that have many options are just too hard for hand coders.

The experimental 801 does not have load-multiple or store-multiple instructions. It uses a subroutine to save and restore registers. The save routine consists of a series of stores into consecutive words and can be entered at any point so that only the required registers are saved. From a performance point of view, this is similar to the situation that exists on System/370 with store- and load-multiple instructions. Those instructions usually have a fixed overhead for start-up that is roughly proportional to the BAL and BR required in the subroutine prologue. In addition, it is possible to bundle other functions related to linkage in the register-save routine. Bumping the stack pointer and testing for stack overflow are examples.

The 801 has a simple data flow. The following two examples show that it is still possible to have powerful instructions by capitalizing on the available hardware.

Load and store instructions have an update or auto-increment form. The idea behind this is that once the CPU has computed the effective address of a load or a store, the updated effective address can be returned to the register file. Such instructions are particularly useful in loops that traverse arrays or strings, and no new hardware is required. On other computers, with auto-increment, pre- and post-increment forms are sometimes provided. That is, the addition is done either before or after the address is sent to the memory bus. Post-increment requires another set of wires to send the contents of the base register directly to the memory. This additional hardware does not seem to be worth the slight convenience of alternative formats.

Storage-to-storage move consumes a lot of computer time. A reasonable guess on System/370 is 2 percent of the executions and 10 percent or more of the time, if MVC and MVCL are both included. On the experimental 801, short moves are done with in-line loads and stores, but long moves require a subroutine. If both the source and target are aligned on a word boundary, which is quite common, a table consisting of a series of update loads and stores can move the data at the speed of the memory bus. Unaligned moves can be done at the same rate by exploiting the shifter-rotater. The idea is to introduce a version of the store instruction that takes the quantity to be stored and rotates it by the difference in alignment between the source and target. Bits shifted out on the right are saved in an internal CPU register for the next store; those saved from the previous store are combined with the quantity to be stored. Thus the central part of the storage-to-storage move loop consists entirely of a sequence of update load and rotate-update-store instructions, and this sequence can proceed as fast as the bus. Of course, this requires a number of instructions to initialize the loop. The existence of shift-and-rotate-type instructions provides most of the functionality to implement this otherwise exotic instruction.

The suitability of the experimental 801 as a target for compilers was a primary consideration.

A target for compilers

The suitability of the experimental 801 as a target for compilers was a primary consideration. On the surface, one might expect that compiler writers would want a machine that was close to the high-level language. However, a machine with basic instructions is better suited to optimization techniques. The most important are code motion out of loops and the elimination of redundant computations. Suppose a program contains a reference to $x(i, j)$. If optimization is done separately on each of the addressing components, the fact that one of the components cannot be optimized need not affect the others. If the above reference occurs in a do loop on i, the load of base of the variable x as well as the load and multiply of j may be moved out of the loop. Newer languages are characterized by complex addressing paths, often involving descriptors. By producing the straightforward code to access such variables and then performing standard optimizations, the compiler can produce good object code. If instructions are complex, any variation in one of the operands inhibits the movement or elimination of the instruction. This subject is covered more fully in References 12 and 14.

A number of details make the 801 approach more effective as a target for compilers. *Condition code* has always been troublesome. To reflect the source languages, one might like to have a relational operator that produces a zero or a one in a register. The problem is that compare is basically a subtract, and construction of the boolean value has to be done very late in the cycle. There is a real danger that the materialization of zero or one in a register may lengthen the basic ALU cycle. Thus it seems best to retain the condition register, which is well suited to its primary use in branching. It also lets one avoid repeating compares—if a three-way choice is made among high, low, and equal—as in a binary search. It is especially important to restrict the instructions that set the condition code. If loads and stores set the condition code, it may be very difficult to insert register-spill code. The scheduling of loads and execute branches can also be constrained by instructions that set the condition register. Finally, it is important that it be possible to easily fetch and restore the contents of the condition register.

It is very desirable to have a set of immediate instructions that contain the operand rather than fetching it from a register. It is not necessary to provide full 32-bit versions, because most constants that actually occur can be defined in 16 bits. The CPU extends these short constants to a full 32 bits by using zeros, ones, or sign bits. If a constant cannot be represented in 32 bits, it has to be loaded from storage or manufactured at execution time. The latter is preferable, and, for this purpose, it is desirable to have a version of load address that shifts the immediate value left by 16 bits. Immediate instructions are heavily used.

The 801 has base-plus-signed-displacement and base-plus-index forms of loads and stores. Studies have shown that it is rare on System/370 to use the base, index, and displacement, all in one instruction. Thus the experimental 801 is not designed with hardware implementing a three-input adder that is rarely used.

Local program addressability should be relative. Perhaps 2 percent of System/370-executed instructions are concerned with establishing, saving, or restoring addressability to the program. An adequate set of immediate operations makes it unnecessary to have an instruction relative-addressing mode for data.

Many high-level languages have rules that can be policed only at run time. Because enforcement is normally costly, most compilers make such checking optional, and it is customary to do without checking during production. This situation has been likened to that of a sailor who uses his life vest only during drills, going without it during the hurricane. To make software checking more economical, the 801 provides instructions that compare the contents of a register with an immediate value or another register. A trap is taken if the test is satisfied. These trap operations can be subjected to the same sort of optimization as is applied to other computations. Thus the number of trap instructions can be reduced, and, when they must be executed, the cost is about the same as that of a register add. Traps are one of the ways that a low-level machine can encourage high-level languages and such good software engineering practice as run-time checks during production.

From the compiler writer's point of view, the experimental 801 is attractive because it is regular. Many decisions are simplified because it always pays to replace two register operations with one. However, it is difficult to obtain an objective measure of regularity. For example, the 801 has only three of the 16 possible boolean operations. This is an irregularity, but it is simply not worthwhile to provide them all, because most are seldom used and the compiler can construct them when they occur using, at most, two instructions. The 801 is a good target for compilers because most of the computations implied by high-level language constructs are variations on addressing code. In practice, computers spend most of their time locating data which in turn locate other data. Such computation is facilitated by full 32-bit addressing, many registers, and instructions that can leave the operands intact. The key to efficiency seems to be reuse. Efficient subroutine linkage is also of utmost importance.

Future uses of hardware

How should computer architects respond to declining hardware costs? From one point of view, the experimental 801 is an expensive machine. It has a full 32-bit single-cycle ALU and shifter. There are 32 general-purpose registers, each 32 bits wide, with three output and two input ports to the register file. Internal buses are all 32 bits wide. Externally, the 801 presents 32 address bits and data bits to the memory-management unit. These are all expensive items and many have been left out of machines with more complex instruction sets.

Because all instructions are 32 bits, the object code for an 801 program is sometimes a little larger than that of System/370. This shows up most on small procedures that do not use 32 registers. An average code expansion seems to be about 20 percent more than System/370 for the same high-level language program. In most cases, this is not significant because data misses in the storage hierarchy are much more frequent than instruction misses. If misses are a problem, the size of the instruction cache can be increased, which is a good use for hardware.

One of the lessons of the 801 experiment may be that the best way to implement a large system is to concentrate on the simpler instructions.

In the future, the largest improvements in performance will probably come about through improving the memory hierarchy. On-chip caches and TLB are an obvious way to speed execution, because they attack the fundamental problem of the performance of the storage system. Wider data paths to memory might also improve performance. For example, the ability to load and store two registers might reduce the overhead associated with call and return, if there were a 64-bit-wide path to memory. Besides work on CPU architecture, the 801 effort at Yorktown has also explored a new memory hierarchy.[15] This is not the place to describe this work, but it is interesting to note some of its characteristics because it sheds light on the uses of hardware. Basically, it is an attempt to share files that are directly mapped into a user's virtual address space. Because the 32-bit address space is not enough, segment registers are introduced in the memory-management unit. Because sharing at the page level would result in too many deadlocks, 16 lock bits are provided for each TLB entry. This permits locking, journaling, and recovery on each 128 bytes of storage in a file. A lot of hardware is required to implement this. The Romp CPU is smaller than the memory-management unit. This cost is justified because the functions performed are very important, and a software solution would require a great deal of overhead. Notice, though, that the hardware does not have to do the entire job. It is enough to give an interrupt on reference to a locked page; systems software can do the rest, as is the case with virtual memory.

One of the lessons of the 801 experiment may be that the best way to implement a large system is to concentrate on the simpler instructions. Thus, there are proposed implementations of more complex architectures that hardwire all the simple, frequently occurring operations and trap on the rest—that is, implement them in software. The success of such an approach has not been proved and will depend partly on human coders changing their habits to treat future systems more like an 801.

Increased parallelism is an attractive way to circumvent the constraints imposed by current technology.

Increased parallelism is an attractive way to circumvent the constraints imposed by current technology. The introduction of an asynchronous floating-point coprocessor undoubtedly speeds many scientific applications. The frequency of floating-point arithmetic in such applications, the increased performance from specially designed hardware, and the ability to execute in parallel provides ample justification. Combining many simple 801s or similar processors may prove to be an efficient way to achieve large-scale parallelism. A word of caution should be raised here, as there have been many failed or only marginally successful attempts to do this. The problem lies with the software, not the hardware.

Concluding remarks

The basic 801 approach was conceived by John Cocke; however, one can trace the notions much further back. In 1951, at the dawn of the computer age, Alan Turing suggested to Christopher Strachey, then a mathematics teacher at Harrow School, that it would be an interesting exercise to simulate one computer on another. Strachey duly wrote a program to simulate the Manchester ACE computer on itself. After a night of debugging he was able to demonstrate that his simulator was able to execute the program that played *God Save The King* on the hooter, albeit very slowly.[16] This was an early graphic demonstration that all computers are logically equivalent. It was certainly not an accident that the problem was suggested by Turing, who had mathematically demonstrated the equivalence of all computers fifteen years earlier. If all computers are logically equivalent, on what basis can the architect make sensible choices when designing the instruction set interface that is implemented by engineers and seen by programmers?

The development of new computer architectures has been driven by many factors. Hardware cost, performance, and reliability have always been important considerations, but other factors have also been taken into account from the very beginning. In 1947, John Mauchly wrote about EDVAC, "A decision must be made as to which operations shall be built in and which are to be coded into instructions.... Ultimate choice must depend upon the analysis by the designer of the character of the work to be performed by the machine, the frequency of the occurrence of operations, and the ease with which the non-built-in operations can be compounded from those which are built in."[17] The 801 emphasis on simple instructions is just a restatement of this old wisdom. Programming has long been recognized as a bottleneck. After all, Turing hired Strachey on the basis of his ability to check out a large program in a single overnight session. Apart from hiring talented programmers, what can be done about programming? The issue is truly complex, because today most programming is done in high-level language. Thus, the exact nature of the computer should concern only the compiler writers, but programmers will use a high-level language only if the compiler produces object code of adequate quality. How diligently must the compiler writer work? What is "adequate?"

The 801 project experiment can be viewed as an attempt to answer these questions in the light of current technology. The microcode approach arose in IBM because it seemed necessary to produce a uniform product line. In 1963, there were many different architectures in the product line, each with exotic features geared to the perceived needs of particular users. Read-only storage provided the means to implement a compatible family of machines with a wide range of cost, while at the same time retaining the same sort of instructions. Increased reliance on high-level language programming tended to undercut the argument for high-level operations based on ease of programming. Also, the introduction of caches provided the opportunity to trade the microstore—which was available only to microcoders—for an instruction cache available to all. The existence of trace tapes showing actual instruction execution demonstrated the importance of hardwiring the simple operations and cast doubt on the economic value of complex operations.

There is no single novelty among the 801/RISC concepts. If anything, they reflect enduring values that clearly go back to the first computers, incorporating the few great ideas that have been developed since then, which include virtual memory and caches. As a research vehicle, the 801 experiment has served as a reminder that hardware is never free, that simplicity is sometimes best, and that a fresh look at existing ideas such as virtual memory and caches can repay big dividends.

Cited references

1. D. A. Paterson, "Microprogramming," *Scientific American* **243**, No. 3, 36–43 (March 1983).
2. M. Auslander and M. E. Hopkins, "An overview of the PL.8 compiler," *ACM SIGPLAN Notices* **17**, No. 26, 22–31 (June 1982); *Proceedings of the SIGPLAN '82 Symposium on Compiler Writing*, Boston, MA, June 23–25, 1982; published by the Association for Computing Machinery, 11 West 42nd Street, New York, NY 10036.
3. G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation via coloring," *Computer Languages* **6**, 45–57 (1981).
4. G. J. Chaitin, "Register allocation and spilling via graph coloring," *ACM SIGPLAN Notices* **17**, No. 26, 98–105 (June 1982); *Proceedings of the SIGPLAN '82 Symposium on Compiler Writing*, Boston, MA, June 23–25, 1982; published by the Association for Computing Machinery, 11 West 42nd Street, New York, NY 10036.
5. V. Markstein, J. Cocke, and P. Markstein, "Optimization of range checking," *ACM SIGPLAN Notices* **17**, No. 26, 114–119 (June 1982); *Proceedings of the SIGPLAN '82 Symposium on Compiler Writing*, Boston, MA, June 23–25, 1982; published by the Association for Computing Machinery, 11 West 42nd Street, New York, NY 10036. See also "The IBM RT PC Subroutine Linkage Conventions," *IBM RT Personal Computer Technology*, pp. 131–133; SA23-1057 (1986), available through IBM branch offices.
6. J. Hennessy, N. Jouppi, F. Basket, T. Gross, and J. Gill, "Hardware/software tradeoffs for increased performance," *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, March 1–3, 1982, pp. 2–11; published by the Association for Computing Machinery, 11 West 42nd Street, New York, NY 10036.
7. D. A. Paterson, "Reduced instruction set computers," *Communications of the ACM* **28**, No. 1, 8–21 (January 1985).
8. A. J. Smith, "Cache memories," *ACM Computing Surveys* **14**, No. 3, 473–530 (September 1982).
9. G. Radin, "The 801 minicomputer," *IBM Journal of Research and Development* **27**, No. 3, 237–246 (May 1983).
10. G. Radin, "The 801 minicomputer," *SIGARCH Computer Architecture News* **10**, No. 2, 39–47 (March 1982).
11. M. E. Hopkins, "A definition of RISC," *Proceedings of the International Workshop on High-Level Computer Architecture*, Los Angeles, CA, May 21–25, 1984, pp. 3.8–3.11; published by the University of Maryland, Department of Computer Science, College Park, MD 20742.
12. M. E. Hopkins, "Compiling for the RT PC Romp," *IBM RT Personal Computer Technology*, pp. 76–82; SA23-1057 (1986), available through IBM branch offices.
13. D. E. Knuth, "An empirical study of FORTRAN programs," *Software Practice*, Vol. 1, John Wiley & Sons, Inc., New York (1971), pp. 105–133.
14. M. E. Hopkins, "Compiling high level function on low level machines," *Proceedings of the IEEE International Conference on Computer Design*, Portchester, NY, October 31–November 3, 1983, pp. 617–619; published by the IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910 (1983).
15. P. D. Hester, R. O. Simpson, and A. Chang, "The IBM RT Romp and memory management unit architecture," *IBM RT Personal Computer Technology*, pp. 48–56; SA23-1057 (1986), available through IBM branch offices.
16. A. Hodges, *Alan Turing: The Enigma*, Simon & Schuster, Inc., New York (1983), p. 447.
17. J. W. Mauchly, "Preparation of problems for EDVAC-type machines," *The Origins of Digital Computers, Selected Papers*, B. Randell, Editor, Springer-Verlag, New York (1973), pp. 365–369.

Martin E. Hopkins *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.* Martin Hopkins is manager of compilers in the Advanced Minicomputer Department. He spent ten years with the Computer Usage Company, working mainly on compiler development. Since joining the Research Division in 1969, he has worked on computer architecture as well as compiler development. In 1985, he received a corporate award for his work on the PL.8 language and compiler. He has a B.A. in philosophy from Amherst College.

Reprint Order No. G321-5289.