Design considerations for IBM Personal Computer Professional FORTRAN, an optimizing compiler

by M. L. Roberts
P. D. Griffiths

An optimizing FORTRAN compiler with power to handle large applications at execution speeds comparable to those of large computers has been implemented on the IBM Personal Computer. This implementation is described, with emphasis on the design decisions that were considered in the development of the compiler.

It comes as no surprise that program execution speed was *the* primary consideration for the designers of the first FORTRAN. What is surprising is the reason for their concern. They were sure that experienced programmers would simply reject out of hand any language whose programs were substantially slower than those the programmers were already writing in machine and assembly language. "We were convinced," wrote John Backus, the leader of the original FORTRAN design group, "that the kind of system we had in mind would be widely used only if we could demonstrate that it would produce programs almost as efficient as hand-coded ones."[1]

Convincing programmers to use high-level languages is no longer a problem, but today, thirty years after Backus' team began its work, execution speed is still a central concern for FORTRAN. Because of the nature of most FORTRAN programs—number-crunching applications used in engineering and scientific work—almost all FORTRAN designers have to wrestle with the challenge of building a compiler that produces code that can be executed at high speeds. (Readers not familiar with some of the basic terms and concepts used here should see the Appendix for definitions.)

Execution speed was an especially critical issue for us in developing a version of FORTRAN for the IBM Personal Computer (IBM PC). Our task was to build a language that would make the IBM PC a powerful engineering and scientific workstation capable of handling mainframe-level FORTRAN applications.

To be able to support existing FORTRAN applications for large machines, and to be able to create new applications with the range and complexity of mainframe applications, the FORTRAN we designed for the IBM PC had to meet two requirements in terms of language features. It had to be a complete implementation of the FORTRAN-77 standard, and it had to incorporate the language extensions popular with FORTRAN programmers.

Building such a full-featured language for a comparatively small computer presents a significant challenge by itself. But it was also essential to design the compiler so that programs written with it could be executed with speeds comparable to those attainable on mainframes, else users would have little incentive to use the smaller machine.

Our task was helped considerably by IBM's decision to make available a coprocessor to handle floating

point arithmetic efficiently. But even with the coprocessor, it was necessary to build into the compiler a wide range of optimizing techniques and features to meet the goal of increased execution speed.

The machine-independent optimizations we incorporated into the final product—IBM Personal Computer Professional FORTRAN (also available from Ryan-McFarland Corporation as RM/FORTRAN)[2]—are all fairly standard. We have been building them into sophisticated FORTRANs for more than twenty years, and they are exhaustively documented in compiler literature.[3] Therefore, this paper will focus primarily on some less well known optimizations—those that are machine- or architecture-dependent—and the way these optimizations were affected by the register design, memory limitations, and floating point arithmetic of the IBM PC.

Machine-independent optimizations

These techniques are designated as machine-independent not because they are totally divorced from the architecture of the processor involved (in many cases they are not), but because they focus on the code a FORTRAN programmer creates. In general, these techniques reorganize the structure and elements of a programmer's code when it is being compiled, so that it can be executed with increased efficiency. This code manipulation basically involves reducing the amount of code the computer must handle at execution time. It trades increased activity and slower speeds at compilation time for faster execution speeds.

The machine-independent optimizations IBM Professional FORTRAN utilizes include the following:

- *Common subexpression elimination.* Common subexpressions are remembered within basic blocks and not recalculated at each use. With the use of this technique, expressions such as

```
A = B + C * D
 . . .
X = C * D/Y
```

  are compiled as

```
t = C * D
A = B * t
 . . .
X = t/Y
```

- *Register remembering.* Current contents of machine registers are remembered, where possible, to eliminate redundant loads and stores. In the preceding example, the value of t would remain in a register as long as possible.
- *Invariant code motion.* Operations whose operands do not change within a DO loop are moved out of the loop. The expression

```
    DO 100 I = 1,10
100 A(I) = X + Y
```

  would be compiled as

```
    t = X + Y
    DO 100 I = 1,10
100 A(I) = t
```

- *Strength reduction.* Expressions involving only DO-loop invariant terms and DO induction variables are calculated using addition rather than multiplication. In the following expression

```
    DO 100 J = 1,10
    DO 100 K = 1,10
100 TWO (J,K) = 0
```

  IBM Professional FORTRAN reduces the calculation of the subscript (J,K) to only 100 additions instead of the 100 multiplications and 100 additions a nonoptimizing compiler would perform.
- *Constant arithmetic.* Constant arithmetic expressions are evaluated at compilation time. The expression

```
A = 1.5 + 3.2
```

  is compiled as

```
A = 4.7
```

- *Constant folding.* Variables known to contain constant values are replaced by those values. The expression

```
J = 2
K = J + 5
```

  is compiled as

```
J = 2
K = 7
```

- Constant terms in subscript expressions are integrated with the array address at compilation time.
- Unnecessary arithmetic or logical operations such as M * 1 or I − 0 are eliminated.
- Conversion of constants from one type to another is performed at compilation time.
- Intrinsic functions are expanded in line, where possible.

Machine-dependent optimizations

Rather than manipulating a programmer's code, machine-dependent techniques involve determining the fastest way to perform a specific operation, given the architecture of a processor. These optimizations require absolute familiarity with the "terrain" of the target processor so that each operation is performed with the highest possible efficiency. The intent of machine-dependent optimizations is figuratively to squeeze a microprocessor, or "chip," for every possible degree of speed by using all its features to optimum advantage and avoiding idiosyncrasies that might slow execution.

One key to designing an optimizing compiler is to make very efficient use of registers.

Registers

One key to designing an optimizing compiler is to make very efficient use of registers. It takes a comparatively long period of time for the computer to retrieve data from memory as opposed to retrieving them from registers. Therefore, we strive to maintain "register residency" for values used most often in a user's program.

A relatively large number of registers are available on the Intel 8088 microprocessor,[4] the processor of the IBM PC—14 or 18, depending on how four of them are used. Although in many respects this is an advantage, the registers of the IBM PC also present a challenge, because in most respects each register is unique. All the registers simply do not have the same functional capabilities: Some are faster than others in performing certain functions, and some simply cannot perform certain other functions at all.

From a compiler designer's point of view, most of the registers in an "ideal" processor would be identical, as they are in IBM's 370/30XX/43XX architecture. With this kind of architecture, a compiler can simply select any available register when one is required. For the processor of the IBM PC, in contrast, the optimizing compiler had to be designed to check at almost every operation to see whether the optimum register is available and if not, to determine whether it is cost-effective to make it available. Figure 1 shows a comparison of the registers in the architectures of the IBM PC and the 370-type computers.

A loop instruction, for example, is best handled only by the Count Register, CX, of the IBM PC. The compiler goes to a great deal of trouble to make sure CX is open when the user program is executing the innermost DO loop of a given algorithm. Design complexity is compounded because CX is also the best register for handling the repeat instructions required for character manipulations. As a result, when the compiler runs into a character concatenation in the middle of a DO loop, optimization may be handcuffed by two routines competing for the same register.

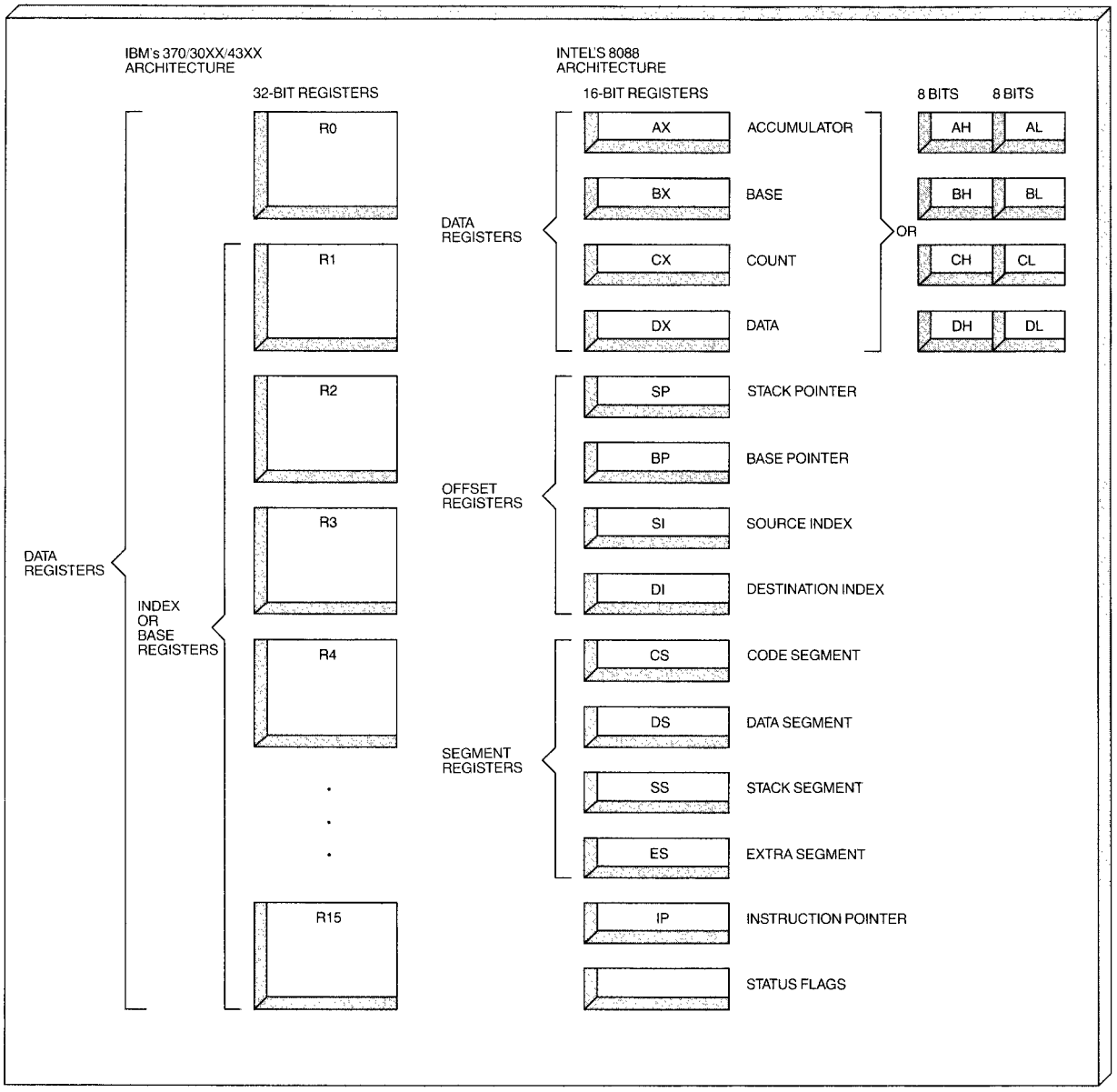
In cases like this, the compromise is generally to use CX for the innermost routine—in this example it would be the character manipulation—and to use another, less efficient, register to control the DO loop. This yields results slower than those attainable if a second, equally fast register were open, but faster than what would occur if CX did not handle the innermost routine, and much faster than what would result if the compiler design had not considered which register should handle which routine.

Another example of the way in which the registers of the 8088 vary is that only one data register, BX, can be used as an index register to calculate the address of an array element. However, commonly occurring index expressions which involve a multiplication cannot be handled directly by BX. They must be calculated first in AX and DX and then moved to BX. Although all four general-purpose registers, AX, BX, CX, and DX, can perform integer arithmetic, only AX and DX can do multiplication and division.

Besides BX, three other registers can be used as index registers: Source Index (SI), Destination Index (DI), and Base Pointer (BP). In many instances these same four can be used as base registers. But when two of these registers have to be used at the same time, there are awkward limitations.

For example, to subscript an item in an array requires both a base and an index. The architecture of

Figure 1 Comparison of registers in IBM's mainframe and Personal Computer architectures



the 8088 requires that one of the two must be BP or BX, and the other one must be SI or DI. Because of another of our optimizations (discussed later), the compiler permanently dedicates BP to addressing elements in the stack segment. Therefore, in complicated subscripting and other operations requiring two index registers, the compiler must make sure BX is available. If it is not, an extra move will have to be generated to free BX. The extra move costs time, but less than it would cost the compiler to use BP and find another register to address the stack.

Limitations of 64K-byte segments

Another challenge presented to the designer of a mainframe-level optimizing compiler by the 8088

microprocessor is the maximum allowable size of segments, which are separate areas in memory where data and instructions are stored.

The processor is actually capable of addressing 1M (1 048 576) bytes of memory. Normally a processor with 16-bit registers would be able to address only 2^{16} or 64K bytes of memory. The 8088 overcomes this inherent limitation by generating a 20-bit address for the beginning of segments. It does this by using the 16-bit segment register as if it had four zeros at its right side, which has the effect of adding four more bits to the register. Figure 2 illustrates this concept.

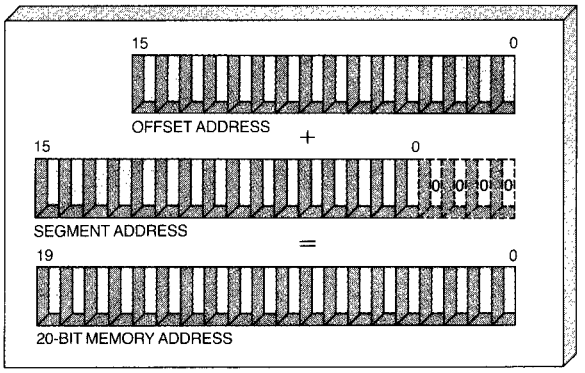
While this increases the amount of memory that can be used by the 8088, obviously an essential task, it also adds to compiler complexity. The processor requires two components to calculate memory addresses: a segment address to indicate the beginning of the segment in which an item resides in memory, and an offset address to indicate how far from the beginning of the segment an element actually resides. To find an element, the processor adds together the contents of the two registers holding these addresses, first appending four zeros to the value in the segment address register.

Having to keep track of two address components for each data element means more work for the compiler. The 8088 minimizes this complexity by assuming that a specific register automatically carries the address of a specific segment, requiring only that an offset be stipulated for an item to be retrieved from memory. If, for example, the compiler assigns the DI Register as the offset address of an item, the processor automatically adds the contents of the DI Register to the contents of the Data Segment (DS) Register. Similarly, the processor assumes that any item with an offset address in the BP Register will have its base in the Stack Segment (SS) Register.

Registers other than the assumed ones can be used to hold base addresses, but to do so requires a segment override instruction. These instructions are costly for an optimizing compiler because each takes about two clock cycles of execution time and one byte of code space. The compiler permanently dedicates BP to addressing elements in the stack, as we mentioned earlier, to avoid just such a segment override.

Another drawback is much more significant. Whereas the base address is in effect 20 bits long, the offset address is only 16 bits. That means that the offset can normally reference items that are no more than 64K bytes (2^{16} bytes) from the base. This restriction limits segment size to 64K bytes, not a tolerable limit for serious FORTRAN programs.

Figure 2 Expansion of 64K-byte addressing limitation



The 8088 processor keeps data and program instructions in separate areas of memory called segments. Up to four segments can be addressed at any one time. The Code Segment holds the program that is currently being executed, the Data Segment holds data appropriate to the program currently being executed, and the Stack Segment holds temporary data and addresses. The Extra Segment (ES) is an alternative Data Segment used in string operations.

The 64K limitation does not impact either the Code or Stack Segment in any significant way. Most adherents of structured programming agree that to be manageable, program modules should not exceed 200 to 300 lines of code. At 20 bytes of object code per source line, which is about twice the number of bytes IBM Professional FORTRAN typically generates, a 64K-byte code segment allows a module of about 3300 lines. Even at 50 bytes per source line, the segment can accommodate a module of 1300 lines, which is at least four times larger than an effective module should be.

Keep in mind also that while individual code segments may be no longer than 64K bytes, the processor imposes no limits on the number of segments that can be linked together and executed. By making every called subroutine a separate segment, the compiler experiences no limit on the amount of code it can handle (though the current version of the IBM Personal Computer Disk Operating System, 3.0, cannot handle more than 640K bytes of total memory).

Stack Segments similarly are not affected by the 64K limitation. Transfers into and out of the operating system, for example, require only a minimal amount of stack space, certainly no more than 256 bytes. With respect to subroutines, the compiler assumes that all calls are made in a vertical line, calculates the maximum amount of stack space each requires, and sums their stack requirements when the subroutines are linked. This final figure is extremely conservative, since calls are rarely made all in a vertical line. The compiler in fact provides an option for the programmer to override this figure and have the linker assign less memory to the Stack Segment.

The compiler divides the logical data segment into physical data segments.

Though the limitation does not affect the Code or Stack Segment, the *Data* Segment for a mainframe-level FORTRAN must be able to handle considerably more than 64K bytes. Almost any serious FORTRAN application involves substantially more than 64K bytes of data.

This limitation creates two different problems to be solved. The easier of the two is how to handle logical data segments which themselves are greater than 64K bytes but contain no individual data items greater than 64K bytes. A "logical data segment" comprises all the local data in a single program unit.

In this case the compiler divides the logical data segment into what we call physical data segments, each of which is smaller than the limitation imposed by the processor. It was possible to allow the physical segments to vary in size considerably, as long as they remained under 64K, but we found there was nothing to be gained by varying the size. We chose to make them as large as possible within the given 64K restraint because by keeping the segments large, we would keep the number needed smaller.

At execution time the linker takes segments that share the same name and class and assigns them to the same area in memory. In this way it is possible to guarantee that separate physical segments will be contiguous in memory when the program is executing.

Once the less-than-64K physical segments are contiguous, addressing a local data item is relatively straightforward. The compiler determines whether it is within 64K bytes of the current segment address in the DS register. If so, it uses DS to access the item. If not, it points ES at the data item (a move that costs at least two instructions because ES cannot be loaded directly from memory).

Here again the compiler does some optimizing. It does not point ES directly at the piece of data; instead it performs a number of calculations to determine the most efficient place to point ES. For example, if the logical segment is about 128K bytes, the most efficient place for ES is right at the 64K boundary. That way the entire local data area or logical data segment can be covered without further change to ES.

The compiler also takes into account whether the data are common or local. (A common data block comprises data shared by more than one of the subroutines in a program unit; local data are used by only one.) DS is not available to point to common data because it is dedicated to local data, so if the data element being accessed is in common, the compiler must use ES. It points ES at the beginning of the block if the element is within 64K bytes of the beginning of the block. If not, and if the element falls within the last 64K bytes of the common block, the compiler points ES 64K bytes prior to the end of the block. In subsequent operations, ES is more likely to be useful at either of these two positions, rather than pointing directly at the data item itself.

Figure 3 shows an example of this optimization to determine the most efficient memory address at which to point a segment register. If an item to be accessed occurs within the first 64K bytes of a common data block (address A + x in the example), the compiler points ES at the start of the data block. If it occurs in the last 64K of a common data block (address N + z), it points ES at the start of the last 64K of that data block. ES is more likely to remain unchanged in subsequent operations if it is pointing at the beginning of 64K-byte segments. Only if the item does not occur in either of these two locations

will the compiler point ES directly at the item itself (address B + y). Analysis showed that it was not cost-effective for the compiler to make more than two attempts to locate the beginning of the 64K-byte segment in which a data item occurs.

The second and more difficult problem the 64K restriction creates is how to address a data element, an array, that is larger than 64K bytes. For mainframe-level FORTRANS, 64K bytes represents an intolerable restriction on array size. It will accommodate a real array of only about 100 × 150 elements, and if double-precision numbers are involved, the array can be only about half that size.

The limiting factor, as mentioned earlier, is that the offset address is normally stored in a single 16-bit register. The compiler erases this restriction by using two 16-bit registers to store a 20-bit offset address: One register holds the 16 low-order bits of the address, the other holds the four high-order bits (with 12 zeros in the leftmost positions).

The segment address is similarly stored in a 20-bit address in two 16-bit registers. Although the compiler might have used the implicit 20-bit addressing of the processor for storing the segment address of some arrays, the starting addresses of all dummy arrays (arrays passed to subroutines) must be calculated at execution time. To reduce the complexity of the compiler, we chose to have the segment addresses of all arrays larger than 64K bytes calculated by the same algorithm at execution time. This decision also gave the compiler a slight increase in efficiency.

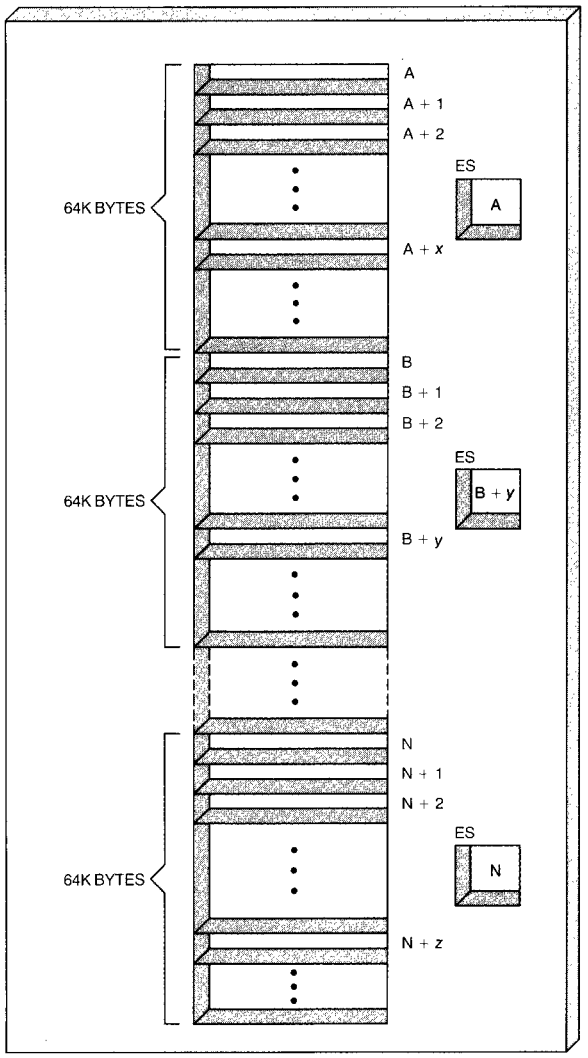
At execution time, the actual 20-bit addresses of the beginning of an array (the array-pointer) and the offset of an item within the array (the subscript-offset) are loaded into 16-bit registers by the following algorithm:

```
MOV BX, array-pointer          load LS portion of array-pointer
MOV AX, array-pointer + 2      load MS portion of array-pointer
ADD BX, subscript-offset       add LS portion of subscript-offset
ADC AX, subscript-offset + 2   add with carry MS portion of sub-
                                 script-offset
                               (LS = least significant, MS = most
                                 significant)
```

Once loaded, the content of these registers, which now represents a 20-bit address, can be converted into the standard 16-bit segment-offset address format by the following algorithm:

```
MOV SI,BX       copy LS portion of 20-bit address
AND SI,OFH      make MOD 16 = segment offset
```

Figure 3 Optimization to determine most efficient memory address




```
AND BX,OFFFOH   remove offset
OR  AX,BX       combine with MS portion of 20-bit address
ROR AX,1        divide 20-bit address by 16 = segment number
ROR AX,1        divide 20-bit address by 16 = segment number
ROR AX,1        divide 20-bit address by 16 = segment number
ROR AX,1        divide 20-bit address by 16 = segment number
MOV ES,AX       move segment number to segment register
```

As a result, the array element can now be accessed at ES:SI. Part A of Figure 4 illustrates how this algorithm uses the four rightmost bits to create the offset.

Though efficient in converting two 20-bit actual addresses into the segment-offset format, this code is

clearly not as efficient as the way the computer addresses data in an array smaller than 64K bytes. The compiler uses it only with arrays that do not qualify for the standard, small-array addressing instructions. (The user also has the option of having the compiler assume that no adjustable arrays will be greater than 64K bytes.)

In determining whether an array qualifies for small-array addressing, however, the compiler cannot simply test to see whether the array is less than 64K bytes. Because the computer in effect appends four zeros to the right end of segment registers, segments can begin only at memory addresses that are multiples of 16 (any binary number ending in four zeros is a multiple of 16). For this reason, depending on where it begins in memory, an array that qualifies for small-array addressing may be up to 15 bytes less than the full 64K bytes a 16-bit register can address. Fifteen bytes, however, is hardly a significant reduction in the 65 536 bytes that comprise 64K bytes of memory.

After developing the first technique, we discovered that we could dramatically reduce the amount of code required by the conversion routine if we were willing to forego an additional 240 bytes in determining the size of arrays that must use the conversion routine. By limiting "small" dummy arrays to those no larger than 64K bytes less *255* bytes, we reduced by a third the code in the second algorithm. That code now looks like this:

```
XCHG AH,BH      exchange MS bytes AX and BX
                (BX now contains the 20-bit address MOD 256;
                  see accompanying diagram)
ROR AX,1        divide 20-bit address by 16 = segment number
ROR AX,1        divide 20-bit address by 16 = segment number
ROR AX,1        divide 20-bit address by 16 = segment number
ROR AX,1        divide 20-bit address by 16 = segment number
MOV ES,AX       move segment number to segment register
```

The array element can be accessed at ES:BX. Part B of Figure 4 illustrates how the new algorithm uses the eight rightmost bits to create the offset.

Not only does this change the number of instructions and thus the object code size, a crucial consideration in putting a large language on a small computer, but it speeds up execution time as well. This sequence is almost 40 percent faster than the one it replaced. Again, extending the 64K-byte limitation on dummy arrays by another 255 bytes is insignificant in comparison to the original limitation and especially in comparison to the optimization that results.

IEEE floating point arithmetic

Two different areas of complexity were encountered in incorporating the IEEE floating point arithmetic standard in IBM Professional FORTRAN. One grew out of the standard itself, the other out of the actual implementation of the standard on Intel's 8087 co-

Two different areas of complexity were encountered in incorporating the IEEE floating point arithmetic standard.

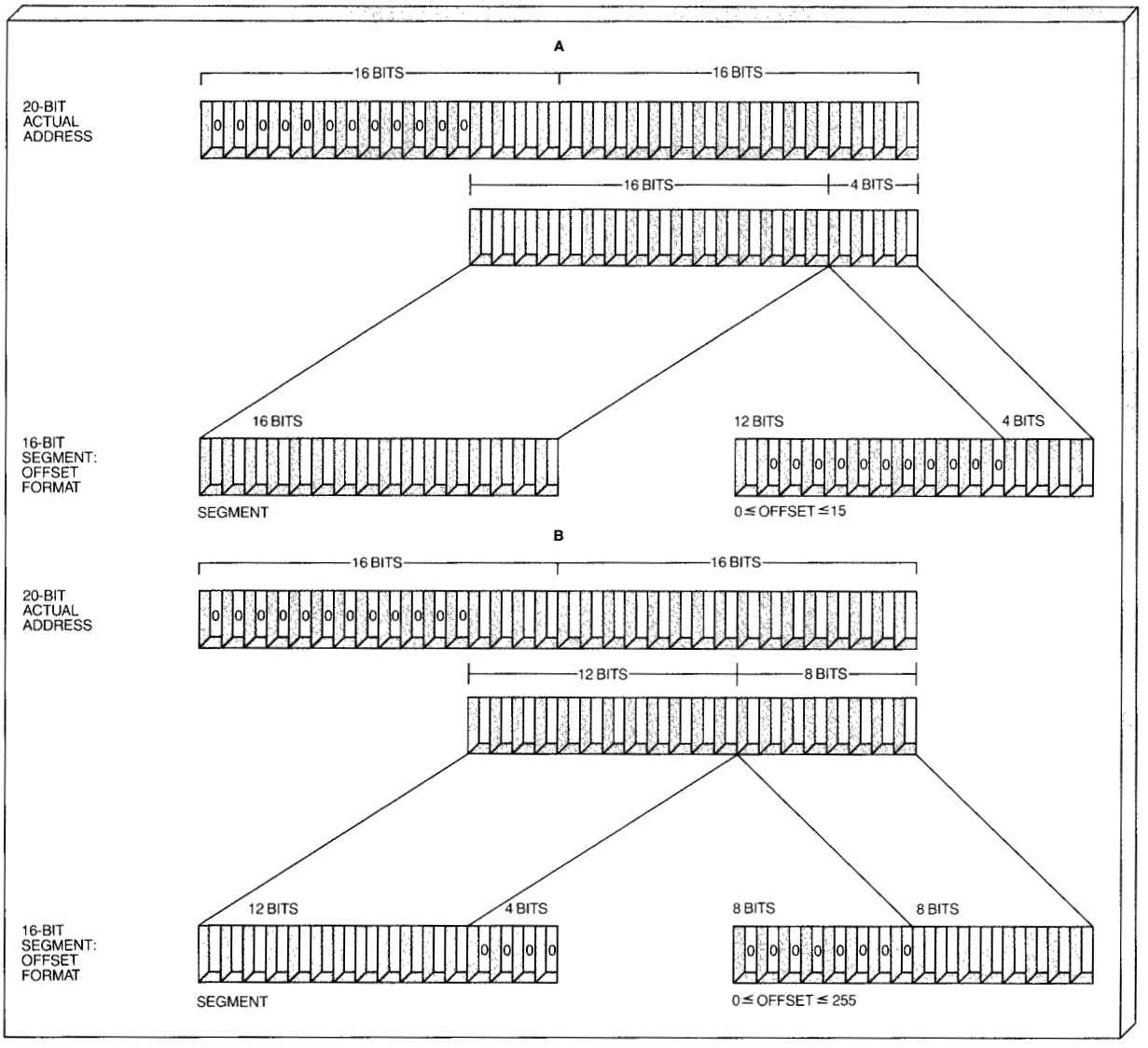
processor.[5] The IEEE standard was a long time in the making, and it appears that some last-minute shifts in orientation by the design group of the standard may have affected Intel's plans. Our design considerations reflect these shifts in a number of ways.

One factor is that the IEEE standard requires the way arithmetic works to be able to vary at the user's option. For example, rounding had to be designed to offer a number of alternatives: a result can be rounded up, rounded down, rounded nearest, or truncated. Arithmetic precision must also be able to be varied in a similar fashion.

The handling of infinity is a good example of the complexities that result from the specific implementation of the standard by the chip. Throughout most of their draft proposals, the IEEE group favored treating infinity as if it were circular. This notion is more useful to numerical analysts than to engineers and scientists, who work with a negative and positive infinity at either end of the number line. Intel built its floating point chip according to IEEE working specifications, where the default mode is a single circular infinity.

To almost everyone's surprise, the IEEE group reversed itself at the last minute and incorporated into the final standard the more traditional plus or minus infinity concept as the default mode. As a result, the default settings on the 8087 chip are contrary to the

Figure 4 Conversion of actual address into segment-offset address



requirements of the standard. Each time the compiler corrects for this discrepancy, the user program loses a little speed because of the extra code it must execute.

An example of how the standard itself causes problems for a FORTRAN compiler is the concept of "Not a Number," usually referred to as NaN. When the result of a computation, such as dividing by zero, has no meaning, the IEEE standard requires that hardware give it a special bit pattern. The bit pattern always propagates itself, no matter what is done to it. The result of any computation involving NaN will always be NaN.

This process clearly works better than the way in which the situation was handled in the past. When an arbitrary number was assigned to represent a computation result having no meaning, there was the very real danger that the results of a subsequent calculation involving it would appear valid.

Nevertheless, NaN poses a problem for a FORTRAN compiler and for most existing FORTRAN programs

because it has not previously been a possible result of a calculation. For example, in the traditional arithmetic IF statement that is still part of many FORTRAN applications, the generated code branches one way if the operand is equal to zero, another way if it is greater than zero, and a third way if it is less than zero. Now there ought to be a fourth branch—if NaN, or, in effect, indeterminate. In these instances we adopt the commonly encountered solution of having the compiler immediately proceed to the next statement when it finds an IF statement involving a NaN.

This difficulty is compounded by Intel's implementation of floating conditional branching. Because of the way the 8087 chip works with the command and control 8088 processor, it is not possible simply to compare to zero and branch. The program cannot branch based on 8087 condition codes. The result of a compare operation must be moved to the 8088, which involves storing the control status. The extra instructions involved cost some time.

Once the status has been moved to the 8086, bits can be tested, and the branch can occur. But the bits that indicate whether a floating point result is greater than, less than, or equal to zero do not map onto the same bits that provide the same information about a nonfloating point result. There are four condition bits that indicate what happened on the floating point chip. When they are moved into the 8088 chip, one of them does not correspond to any 8088 bit. To be tested, it requires a special test bit instruction that sets a condition code. All of this requires more time and increases the complexity of the compiler, because it must have two separate conditional branch generators.

Another area in which the actual implementation of the standard forced a number of design decisions was denormalized numbers. The IEEE group added these numbers to the standard to handle the problem of underflow. In traditional floating point implementations, a number becomes smaller and smaller and then suddenly becomes zero. This gap between the smallest representable number and zero can cause problems in certain applications. To handle them, the IEEE group introduced the concepts of gradual underflow and denormalized numbers. The exponent of these numbers is the smallest possible value, but their mantissa may not be normalized (may not have the high-order bit set on). In the IEEE standard, these are considered valid numbers, which may be used in computations and may in fact cease to be denormalized. When a denormalized number gets smaller, it too will reach a point where it suddenly becomes zero. In theory, however, these numbers provide more accuracy than was available before.

Intel chose to implement denormalized numbers in a way that differs from the standard. When a denormalized number is loaded into a register, it is not loaded as a denormalized number but as what Intel calls an "unnormalized" number. Unnormalized numbers have characteristics different from those of denormalized numbers. In particular, they no longer follow all the IEEE rules for arithmetic. This leads to several interesting quirks; for example, a number divided by itself may not be equal to one. Or if a denormalized number is stored in memory and then reloaded into a register, its value no longer equals its original value. We discovered after extensive analysis that to circumvent Intel's implementation would require a significant performance penalty.

Another set of challenges resulted from Intel's implementation of floating point registers as a true stack. As an element gets pushed farther down the stack, it gets farther from the top of the stack. This causes several problems for an optimizing compiler which needs to keep track of several partial results that will be used more than once. One problem occurs because certain actions can occur at the top of the stack only. For example, storing an element that is not on the top of the stack requires executing an exchange instruction.

The exchange instruction is very fast compared to other floating point instructions, so there is no significant performance penalty in the user's program. On the other hand, it is very complicated for the compiler itself to keep track of what expressions are in what registers. A complex number, for example, consists of two real numbers and therefore requires two different registers to represent it. Tracking these two registers while calculating complex expressions required us to build a very elaborate piece of code.

Another source of difficulty is that the chip keeps track of whether a register has a value in it or not, and it will not allow a new value to be loaded over a current value, as traditional architecture allows. Old values must be removed by additional code. Similarly, there is no convenient way of emptying all the registers at once. The most efficient way to accomplish this is to reinitialize the chip, even though many other items, such as the infinity mode defaults, must also then be reinitialized, again with certain performance penalties.

Our decision on how to do I * 4 (32-bit) arithmetic illustrates some of the conflicting factors that affect the choices made while building an optimizing FORTRAN compiler. The basic question was whether I * 4 arithmetic should be handled on the floating point chip or by instruction sequences on the 8088 chip. Since there is no 32-bit arithmetic capability on the 8088, it must be simulated by using two 16-bit integers.

Our tests showed that while addition and subtraction can be done faster by simulation on the 8088, multiplication and division can be done faster on the floating point chip. We made the design decision not to do half the arithmetic in one place and half in another, and settled on doing all arithmetic on the floating point chip. This offered a number of advantages, perhaps the main one being that it made the conversion back and forth between integers and real numbers very easy.

The trade-off between faster multiplication and division and slower addition and subtraction works on the 8088 chip—much better than it does on the 80286 chip. Although the 80287 floating point chip is about ten percent slower than the 8087 chip (because of a more complicated bus architecture), the 80286 is about five to six times faster than the 8088. This speeds up the addition and subtraction much more significantly than it slows down the multiplication and division. Implementations of IBM Professional FORTRAN specifically for 80286-based machines therefore handle I * 4 arithmetic on the control chip rather than on the floating point chip. This is a good example of how optimizations for a specific processor may not work in the same way for all members of a chip family.

Concluding remarks

Writing an optimizing compiler may be a complex undertaking. Testing its effectiveness is not. It is simply a matter of measuring the execution speed of its compiled programs and comparing the results to the same programs written in other versions of FORTRAN. Testing with our benchmark programs has shown that IBM Professional FORTRAN (RM/FORTRAN) outperforms, by a very significant margin, nonoptimizing FORTRAN compilers marketed for smaller systems such as the IBM Personal Computer.[6]

The end result of our design work, IBM Professional FORTRAN (RM/FORTRAN), shows clearly that it is possible to build an optimizing FORTRAN compiler small enough to run on a desktop computer system and powerful enough to handle large-machine applications at large-machine execution speeds.

Appendix: Basic definitions

Presented here are some very basic definitions for readers not familiar with compilers and their operation. At its most basic level, a *compiler* is a series of instructions enabling a machine that can understand only zeros and ones (the presence or absence of electrical charges) to perform a sequence of events described in terms human beings find convenient to work with. It does this by taking a program written in a language like COBOL or FORTRAN and translating it into a long series of zeros and ones. When run through a computer during program execution, the translation prompts the computer to perform a sequence of the simple operations it is capable of.

The series of instructions written in a human-oriented computer language like FORTRAN or COBOL is called *source code*. Translation (or compilation) of source code results in *object code*, the zeros and ones that the machine can understand and that are created when the compiler compiles the source code.

Because human logic and problem-solving techniques are vastly different from machine logic and capabilities, designing a program that will automatically translate into executable object code any combination of the elements of even a simple computer language is a highly complex undertaking. It involves using only the simple operations a machine is capable of to perform the following tasks: keeping track of an enormous number of data items; keeping track of the exact locations where all these elements are stored in memory; and keeping track of the sequences in which operations are performed and elements are fetched from memory into temporary storage areas called *registers*, where they can be held, manipulated, and made available for high-speed access by the central processing unit (CPU).

All of this describes even the most basic high-level language compiler on the simplest of computer processors. The complexity increases, almost exponentially, when the language has all the sophistication of a high-powered FORTRAN; when the processors involved are as complicated as Intel's 8088 and 8087 chips, which are at the heart of the IBM PC; and when the aim is not just to produce executable object code, but highly efficient object code that can be executed as quickly as possible.

The key to building a compiler that creates such high-speed object code is a set of techniques called *optimizations*. These are routines embedded in the compiler which examine the source code as it is being compiled and reorganize it so that the number of instructions (object code) that must be executed are minimized and so that the instructions are executed in the fastest way possible.

Cited references and notes

1. J. Backus, "The history of FORTRAN I, II, and III," *ACM SIGPLAN Notices* **13**, No. 8, 167 (August 1978).
2. IBM markets the product for its Personal Computer family members under the name IBM Personal Computer Professional FORTRAN by Ryan-McFarland Corporation. It is an implementation, developed in cooperation with IBM, of Ryan-McFarland's RM/FORTRAN compiler, which was delivered in 1983 for M68000-based computers. The implementation is also marketed by Ryan-McFarland, under the name RM/FORTRAN, for all 8086/88/286 and MS and PC DOS-based systems.
3. For one of a number of bibliographies on the subject of language optimizations, see F. E. Allen, *Bibliography on Program Optimization*, Research Report RC-5767, IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598 (1975).
4. Professional FORTRAN was written specifically for Intel's 8088 and 8086 processors. The two processors differ only in terms of their internal data buses: The 8088 has an 8-bit bus and the 8086 has a 16-bit bus. For the sake of convenience, the paper refers only to the 8088 processor; the reader should note that comments apply equally to the 8086. For more information about these processors, see *The iAPX 86, 88, 186, and 188 User's Manual*, 210911, Intel Corporation (1983); available from the Intel Corporation, 3065 Bowers Avenue, Santa Clara, CA 95051.

   Professional FORTRAN also runs on the IBM Personal Computer AT, the CPU of which is Intel's 80286 microchip. Running in the "real mode," the 80286 appears to be a very fast 8086 processor. An important difference between the two processors is noted at the end of the section on IEEE floating point arithmetic in this paper.
5. IBM Personal Computer Professional FORTRAN (RM/FORTRAN) requires either the IBM Personal Computer AT Math Coprocessor or the Personal Computer Math Coprocessor.
6. Specific information is available from the authors.

Reprint Order No. G321-5237.

Mark L. Roberts *Ryan-McFarland Corporation, 609 Deep Valley Drive, Rolling Hills, California 90274.* Mr. Roberts joined Ryan-McFarland in 1974. He has had various assignments in compiler development and is a Senior Technical Staff Member. For the past four years he has been responsible for the design and development of Ryan-McFarland's optimizing FORTRAN compilers. Mr. Roberts received a B.S. in mathematics from the University of Washington and an M.S. in computer science from the University of California at Los Angeles.

Peter D. Griffiths *Ryan-McFarland Corporation, Crown House, Turners Hill Cheshunt, Waltham Cross, Herts EN8 8NN, England.* Mr. Griffiths is a Senior Technical Staff Member at Ryan-McFarland and is head of the company's London office. He joined the company in 1976. For the last two years he has shared responsibility with Mr. Roberts for the design and development of Ryan-McFarland's optimizing FORTRAN compilers. Mr. Griffiths studied mathematics at Manchester University in England.