Architecture and software support in IBM S/390 Parallel Enterprise Servers for IEEE Floating-Point arithmetic

by P. H. Abbott
D. G. Brush
C. W. Clark III
C. J. Crone
J. R. Ehrman
G. W. Ewart
C. A. Goodrich
M. Hack
J. S. Kapernick
B. J. Minchau
W. C. Shepard
R. M. Smith, Sr.
R. Tallman
S. Walkowiak
A. Watanabe
W. R. White

IEEE Binary Floating-Point is an industry-standard architecture. The IBM System/360™ hexadecimal floating-point architecture predates the IEEE standard and has been carried forward through the System/370™ to current System/390® processors. The growing importance of industry standards and floating-point combined to produce a need for IEEE Floating-Point on System/390. At the same time, customer investment in IBM floating-point had to be preserved. This paper describes the architecture, hardware, and software efforts that combined to produce a conforming implementation of IEEE Floating-Point on System/390 while retaining compatibility with the original IBM architecture.

Introduction

In late 1995, the IBM System/390* Division decided to add support for IEEE Floating-Point arithmetic to the evolving S/390 Parallel Enterprise Servers*. The objective was to address the increasing prevalence of IEEE Floating-Point in new workloads that customers wanted to host on these new machines. This paper describes the resulting hardware architecture and software support provided to allow developers of these new workloads, as well as customer programmers, to employ the new facilities.

IBM System/360* implemented floating-point arithmetic in hardware and offered the feature in all models of the product line, a significant advance at the time. This capability brought the performance of hardware floating-point to customers using even the smallest S/360 machines. The facilities included four floating-point registers (FPRs), short- (32-bit) and long- (64-bit) precision data types, and an innovative hexadecimal representation allowing for compact exponent encoding and minimizing the shifting required to normalize operands. The elements of this architecture were implemented in several ways in the different models of S/360 to provide appropriate levels of performance when compared to other machines in the

0018-8646/99/$5.00 © 1999 IBM

line and with other manufacturers' processors. Later, an extended-precision (128-bit) data type was added to the hardware, providing significant additional precision to floating-point representation and computation. This support was an IBM architectural exclusive at the time of its introduction and remains one today. This architecture has been implemented on all IBM large-scale machines following S/360, up to and including the current S/390 Enterprise Server products.

In the interim, the industry, including IBM, looked for a standard floating-point system that could be implemented efficiently on smaller systems, especially the then-emerging line of RISC machines, and one that addressed deficiencies in existing floating-point systems. This work was conducted through the IEEE Floating-Point Working Group P754. IBM's first "straw-man" architecture proposal for implementing the IEEE Floating-Point draft standard [1–5] in the then-current S/370* architecture dates from 1982. In 1985, ANSI/IEEE Standard 754-1985 [6] was approved. Today, most of the IBM systems with which S/390 Enterprise Servers interoperate implement this standard, including all Personal Computers, all RISC workstations (including RS/6000*), and AS/400*. Programming languages have begun to include IEEE Floating-Point as a required data type.

IBM's current effort began in 1989, when the System Architecture group developed an initial proposal for an IEEE Floating-Point implementation. This architecture effort, augmented by hardware and software engineering activity, continued at varying levels of intensity until the current one began in 1995.

- *S/390 transformation*

S/390* is in the midst of a transformation that began in the early 1990s. The large S/390 mainframe of that time was an extremely capable system. It hosted one or more very robust S/390 operating systems, providing a comprehensive hardware/software system capable of running workloads inconceivable for other platforms, including those of the world's largest banks, manufacturing companies, insurance companies, stock exchanges, and airline reservation systems.

However, this system was perceived to be too costly by both actual and prospective customers. It was also considered proprietary and was labeled a "legacy" system, a pejorative term with no concrete definition but an implied meaning of old or obsolete, to be replaced by inexpensive, modern, open systems based on client/server technology.

The large systems of the time supported a vertical model, in which customer growth requirements were met by providing larger systems featuring faster basic engines configured either as uniprocessors or as multiprocessors containing two to six engines. These machines were built from bipolar technology, which provided the high performance required to support vertical growth. However, this technology was also high-cost and high-risk, requiring special environmental support, including chilled water for cooling. These factors raised the machines' total cost of ownership.

Those systems also featured software technologies and workloads characterized as proprietary or "legacy," such as SNA, centralized OLTP (on-line transaction processing, as delivered through subsystems such as IMS and CICS), and batch processing. The technologies that provided the ability to support these workloads delivered solid value and were critical to the large business organizations that deployed them. They were, however, continuing to diminish in popularity within the industry. These systems continued support for the floating-point system first implemented with S/360, referred to in this paper as hexadecimal floating-point, or HFP.

- *Current environment*

At this point in the S/390 transformation, significant changes have been accomplished. The most important of these has been a change to a horizontal growth model, featuring the S/390 Parallel Sysplex*. Today, when customers need additional capacity, they can grow horizontally by adding an additional system and connecting it to the existing ones. This is accomplished by connecting each of the systems to an additional processor configured as a coupling facility, or CF, a form of electronic shared memory. The CF supports a set of operations that, with the accompanying subsystem software support, provides scalable, low-overhead, single-system-image operation across multiple physical machines, allowing functions such as *N*-way (where *N* is the number of systems) sharing of data managed by DB2* or DL1. Work for the IMS subsystem can be routed to any system in the Parallel Sysplex, providing another single-system view of the environment. The combined system also supports higher availability through facilities provided by the communications subsystems and the coupling facility.

In addition, OS/390*, the integrated MVS system, has embraced open-systems function by providing an implementation of an XPG4-compliant UNIX** kernel and a UNIX shell as an alternate execution environment to the others it supports. This simplifies the porting of UNIX applications and also facilitates the implementation of important middleware technologies on OS/390. Examples of such applications and technologies include SAP R/3 Data Base Server and Lotus Domino**, some of the new workloads enabled for the transformed S/390 Enterprise Server. S/390 is also providing an implementation of the Java** virtual machine [7], a technology facilitated by having a UNIX environment on OS/390. The native-float data type in the Java language [8]

is an IEEE float. The implementation of IEEE Floating-Point hardware on S/390 will enable these new technologies to run on S/390 much more efficiently.

Design objectives

Our basic objective was to add IEEE Floating-Point as an alternative to the existing HFP, not as a replacement for it. A set of design objectives for the project developed, beginning with the previous architecture efforts and continuing as we extended them to provide an architecture and design for the S/390 IEEE Floating-Point hardware and software implementation. These objectives are described below.

- *Affordability*

The first objective was affordability. It led us to provide a full hardware implementation, but also to provide software to support a focused set of new applications and a customer development capability, as opposed to support across the OS/390 system and across the S/390 platforms. This resulted in support in the OS/390 BCP (Basic Control Program), the UNIX kernel, the High-Level Assembler (HLASM), the C/C++ Compiler, the C Run-time Library, the C++ Run-time Library, the Java virtual machine, Language Environment (LE) for OS/390, the *dbx* debugger, and the VisualAge* Remote Debugger, all in the initial release. To allow development of customer applications, we provided support in VM/ESA* for guest virtual machines that use IEEE Floating-Point.

- *Performance*

The second objective was performance. Any additional overhead in context switching created by the new additional floating-point registers (AFPRs) was important to control. We adopted a "pay-as-you-go" philosophy whereby the overhead is incurred only if the hardware feature is actually being used. The floating-point hardware feature must be activated before it is available for customer use. This activation is accomplished on a per-process basis by the first attempt to use the new instructions or registers.

- *Serviceability*

The third objective was serviceability. The FPRs (the existing four as well as the AFPRs) are made available to both existing HFP arithmetic and the new IEEE Binary Floating-Point (BFP) operations. An architecture tradeoff between serviceability and the desire to economize on instruction operation codes was settled in favor of serviceability. This led to a new, separate instruction set for IEEE Floating-Point, instructions visible to a debugger or in a memory dump, as opposed to a mode-set operation and reuse of the existing instruction set.

It is important to note that the new AFPRs are available to both forms of floating-point computation. As such, a dump of the registers means nothing without an understanding of which type of computations have been performed.

- *Compatibility*

The fourth objective was compatibility. It guided us when implementation issues arose, associated with areas of the specification that are not clearly spelled out. The decision made was to remain compatible with RS/6000 and AIX* in such circumstances. This approach facilitates porting applications to OS/390 that originated on UNIX platforms.

- *Definitive support*

The fifth design objective was definitive support. Specifically, this meant that we wanted to avoid issues of imprecision as to whether an environment does or does not support IEEE Floating-Point. The specific choice we made was to provide IEEE Floating-Point support in OS/390 Version 2 Release 6. This led to the development of an instruction simulator, included in the OS/390 Base Control Program (BCP), that provides support for the instructions when the operating system is running on a machine without the hardware feature. The machines providing the necessary hardware support are the G5 and its successors. However, any program compiled to use the IEEE instructions will run on an OS/390 Version 2 Release 6 system, albeit at degraded performance on the G4 and earlier machines.

Another aspect of definitive support was long-term coexistence between HFP and BFP. There is minimal support designed to encourage migration from HFP to BFP over time. Existing applications, if either not recompiled or recompiled without change, continue to use HFP and four FPRs. Such applications are encouraged to use, where possible, the AFPRs, but not to convert to BFP unless its specific advantages apply. Conversion to BFP requires application analysis to ensure that adequate range is available. Mechanically, using either BFP facilities or the AFPRs for HFP requires application recompilation and specific compiler directive-setting.

- *Conformance to the standard*

The sixth objective was conformance to the standard. This goal, taken from the hardware architecture, was to provide a fully conforming hardware implementation of the standard. The target was to put as few dependencies for conformance as possible on the OS/390 BCP. Meeting this objective allows us to enhance the control program independently of the state of the standard and conformance to it, a desirable degree of freedom with

respect to future work. It will allow us to broaden the scope of software support for IEEE Floating-Point in OS/390 in response to market requirements, without depending on enhancements to the control program.

Architecture

The System/360 architecture, introduced in 1964, was the definition of the interface to the machine, both hardware and microcode, as observable by the "program." This was one of the first instances of the use of the term "architecture" pertaining to a computer system. Since that time, the term has become very popular and has acquired diverse meanings. It is used to describe both hardware and software interfaces, as well as system structure. In this paper, the terms "architecture" and "hardware architecture" are used in the original System/360 sense. For S/390, the official term for the current version of this interface is Enterprise Systems Architecture/390* (ESA/390*) [9].

Architecture design process

Architecture design is an iterative process; it involves coordinating requirements from many areas, including hardware, programming, and marketing. These requirements are continually changing and seldom present a consistent picture. This problem is particularly acute for ESA/390, because it is supported by several different operating system platforms (OS/390, VM, TPF), each on different design and delivery schedules.

For any particular architecture facility, it is unusual for the hardware and corresponding software design to coincide; a more typical situation is that the hardware design for the first machine to implement a new architecture facility is frozen before the software design begins. Thus, a good deal of guesswork may be needed to predict requirements before they are known. Typically, new architecture facilities delivered on a machine must first be tolerated, and later exploited, by each software platform. The very nature of the support for IEEE Floating-Point exacerbated this problem.

Unlike some architecture facilities, which are implemented in microcode, the implementation of IEEE Floating-Point in the hardware requires fundamental changes in dataflow. Consequently, the designers needed much more lead time than usual. In fact, the basic architecture model was completed in 1992, and most of the basic dataflow changes were included in the hardware for G4 in preparation for introduction of the facility on G5.

Compatibility

Compatibility, one of the strengths of ESA/390, is also a very important factor in the architecture design process. For ESA/390, compatibility among hardware models is even more important than it is among releases of software. The typical life of a machine in the field is five to ten years. A single customer may have multiple machines with an age span of at least ten years. On the other hand, the same customer is not expected to run more than two releases of a particular operating system, and even then not for an extended period of time. Consequences of incompatible changes are much more serious in the hardware architecture than in the operating system. Occasional compatibility problems between versions of an operating system can be solved by running back levels under VM or LPAR. The corresponding situation for hardware—keeping old machines—is much less palatable.

Compatibility is of particular concern in the case of IEEE Floating-Point, as this facility is intended to be used by application programs running in the problem state. In the ESA/390 environment, application programs have learned to enjoy complete bit-for-bit compatibility across the entire product line for more than three decades, a trend that is expected to continue for several more.

Only one chance to get it right

The original System/360 architecture for HFP did not define a guard digit for long operands. Not until after the first machines were shipped was it discovered that this was a serious problem. The hardware was redesigned and the change (primarily in microcode) was retrofitted in the field. With the G5 implementation of IEEE Floating-Point, such a change would require hardware changes in the chip and would be much more expensive than the 1968 alteration. Since this early correction, the System/360 definition of HFP has been unchanged for more than 30 years. Even additions to the HFP architecture have been rare. The extended format and seven operation codes were added in December 1967, and two operation codes for square root were added in September 1991.[1] In short, with IEEE Floating-Point, we had only one chance and would be bound essentially permanently to what we produced.

- *ESA/390 design objectives*

With the foregoing in mind, the following design objectives were used as a basis for developing the hardware architecture [10] for the ESA/390 floating-point (FP) extensions:

- World-class definition for both current and future markets.
- Fully conforming implementation of the IEEE standard (all of the "shalls" and most of the "shoulds").

[1] In addition to the mainstream facilities, a vector facility with 171 operation codes was added in January 1986, and nine new vector operation codes were added in September 1991.

- Full support for both single and double formats.
- Performance matching or exceeding that of the original HFP.
- Support for the extended format, not just for internal computation, but externalized to the application in the same manner as HFP.
- A more complete set of extended-precision operations than previously available for HFP.
- AFPRs to improve the support of complex arithmetic and extended formats.
- Control-program-independent application interface.
- Where practical, BFP additions also provided for HFP.
- Coexistence of HFP and BFP applications and data.

- *Dual-radix floating-point representation*

Support of two floating-point radixes in the same machine presents some special problems. Among these are questions of whether the two should share the same FPRs, share the same exceptions, and share the same operation codes.

Common registers

The same FPRs are used for both BFP and HFP data, because separate sets increase the complexity of the design, the hardware cost, and the overhead for context switching. Further, separate registers do not solve any of the linkage problems associated with mixed mode.

Separate exception indications

Exception handling for BFP is quite different from that for HFP. BFP has five exceptions, with masks and status flags for each; HFP has four exceptions, two masks, and no status flags. Three exceptions are common to BFP and HFP (overflow, underflow, and division by zero), so using the same interruption codes for both modes was a possibility. But even for these three exceptions, the action taken by the machine is not the same, so there was no hardware advantage to using the same interruption codes. It was also expected that in a mixed BFP/HFP environment, different trap-handling routines would be invoked for the BFP and HFP exceptions, so new interruption codes were defined for all five BFP interruption conditions.

BFP/HFP conversion instructions

Coexistence and migration from HFP to BFP are provided by instructions that convert between the HFP long format and the BFP short and long formats. Instructions for HFP short operands are not necessary, as conversion between HFP long and short is trivial. Conversion of extended operands is expected to be too infrequent to justify implementation in hardware.

The BFP/HFP conversion instructions present some unique problems as to whether HFP or BFP exceptions should be reported and whether the BFP rounding modes should apply. It is expected that these instructions will be used in both BFP and HFP environments, so they were defined to be independent of both. Rounding is provided explicitly, independently of the BFP rounding modes, and exceptions are reported by means of the condition code rather than by program exceptions.

Mode bit or separate instructions

A key decision that must be made in such a design is whether to reuse the existing HFP operation codes for BFP, with the run-time interpretation controlled by a mode bit in the PSW, or to introduce a new set of operation codes. A similar choice had to be made in the IEEE standard for the four different styles of rounding. Although the standards committee could have left this choice of implementation open, it decided to design an explicit mode setting that would be available to the program.

Using a mode bit and reusing operation codes offers the clear advantages of simpler hardware and fewer new operation codes. It also allows for mode-independent subroutines (facilitated by a special LOAD MODAL CONSTANT instruction, which loads one of two operands from storage depending on the floating-point mode setting) and might simplify compiler code generation.

While the hardware advantages are real, the software benefits are less compelling. Algorithms for intrinsic functions are entirely different in the two floating-point formats, and the reuse potential of other subroutines is marginal, given the very different mechanisms provided for dealing with exceptions. Since compiler code generation is heavily table-driven, a separate set of operation codes is not a major issue.

A modal approach makes sense in the case of the IEEE rounding style because the different modes apply nearly uniformly to all instructions. In the BFP/HFP case, the underlying instruction sets are somewhat different, which weakens the leverage that might otherwise be possible. Further, the modal approach creates compatibility problems for old programs written for HFP if there is any possibility that they may be executed in the BFP mode. Finally, the modal approach presents a significant difficulty (even for new programs) in being able to diagnose a case in which a program that has been written for one mode has been inappropriately called in the other (resulting invariably in gibberish, which may or may not be quickly detected).

For these reasons, it was decided to pay the price in hardware and use separate operation codes to ensure the ability to produce robust applications.

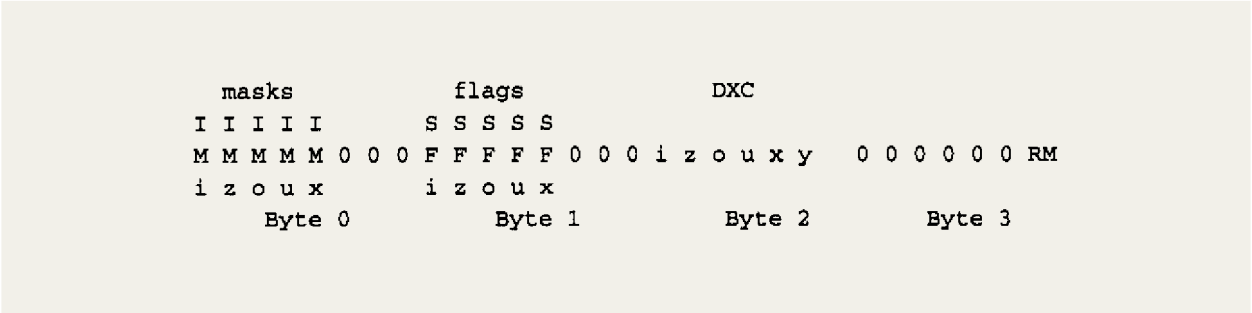

Figure 1

FPC register overview.

Table 1 FPC register bit assignments.

Byte	*Bit(s)*	*Name*	*Abbr.*
0	0	IEEE-invalid-operation mask	IMi
0	1	IEEE-division-by-zero mask	IMz
0	2	IEEE-overflow mask	IMo
0	3	IEEE-underflow mask	IMu
0	4	IEEE-inexact mask	IMx
0	5–7	(Reserved)	0
1	0	IEEE-invalid-operation flag	SFi
1	1	IEEE-division-by-zero flag	SFz
1	2	IEEE-overflow flag	SFo
1	3	IEEE-underflow flag	SFu
1	4	IEEE-inexact flag	SFx
1	5–7	(Reserved)	0
2	0–7	Data-exception code	DXC
3	0–5	(Reserved)	0
3	6–7	Rounding mode	RM

- *Registers*

Several considerations determined the register-related aspects of the architecture.

Floating-point control register

BFP operations require several new control and status bits: five IEEE exception status flags, five IEEE exception trap masks, and two bits to specify any of four rounding modes defined by the standard. The traditional container for this type of information is the PSW, but there is not enough room. This information is therefore placed in a new register, called the floating-point control (FPC) register, which is accessible by the problem program. In addition to the above information, an 8-bit data exception code (DXC) is also placed in the FPC register when an IEEE trap occurs.

An overview of the 32-bit FPC register is shown in **Figure 1**. Details are given in **Tables 1** and **2**.

Additional floating-point registers

The original System/360 architecture provided four 64-bit FPRs. In 1964, four registers constituted an innovation and a not inconsiderable amount of hardware. Intrinsic functions written in assembly language performed quite well with four registers; compilers at that time could exploit only one or two.

Today, four FPRs are too few for optimum performance. The simplest math library routines require four FPRs; this number doubles for complex arithmetic and doubles again for extended precision. With the current technology, the cost to provide additional registers in the hardware is minimal. Support of BFP provided the opportunity to increase the number of FPRs, for both HFP and BFP.

The four original System/360 FPRs were numbered 0, 2, 4, and 6, making the numbering consistent with that for the even–odd general register pairs. Extended-precision operands, which were added later, were accommodated in pairs of registers (0/2 or 4/6), making only two registers available in that format.

The System/360 instruction format provides 4-bit register fields; thus, the bits in the instruction formats have always been there to expand the number of available registers to 16. Coupling register pairs for extended-precision operands was done to provide a compatible superset of the original System/360 implementation. **Table 3** shows the layout of the FPRs, including the coupling of register pairs for extended-precision operands.

Activating additional registers

Bit 13 of control register 0 (CR0.13) controls the use of the AFPRs. If a program attempts to use any of the 12 new floating-point registers or the FPC register when this bit is zero, a program interruption occurs. (Since all BFP instructions depend on the masks and the rounding mode in the FPC register, they implicitly use that register.)

This approach provides two advantages:

1. An application can use the BFP instructions and AFPRs independently of the operating system platform. Thus, a control is required to prevent an application from using the registers if the control program is not prepared to save and restore them. If a new program attempts to use the new registers on a new machine running an old control program, a data exception is reported, and no harm is done.
2. The additional registers need not be saved and restored when running programs that do not use the new registers.

- *Conformance to the standard*

To our knowledge, there is no hardware implementation that conforms to the standard. The standard states that "hardware components that require software support to conform to the standard shall not be said to conform apart from such software." A great deal of effort was expended to provide a fully conforming hardware implementation of the standard. This was accomplished for the basic formats in all but one area: binary–decimal conversion.

Conversions between BFP format and decimal strings

The standard requires conversions between "basic format" floating-point numbers and decimal strings, but does not specify the format of "decimal strings." This is the only area required by the standard and not provided by the hardware. The software conversions are described in the section on conversion between decimal and binary floating-point.

- *BFP formats*

The standard defines two "basic" formats: single and double. ESA/390 provides both, but uses the terms "short" and "long" (for 32-bit and 64-bit formats), as these terms have been used since the inception of System/360 for HFP operands of corresponding lengths.

The standard defines "extended" formats but permits a great deal of implementation flexibility in this area. These formats provide additional exponent range and precision above the widest basic format supported by the machine. They were primarily intended to be used to facilitate intermediate computations by math library routines and were not necessarily meant to be externalized. However, the S/390 implementation does externalize the extended format, so much care was taken in the definition. The ESA/390 extended format is 128 bits and meets or exceeds the requirements of the standard in every respect. The bit representation was chosen to be completely consistent with that used by the standard for single and double; it includes a sign bit, a 15-bit exponent, and a 112-bit fraction with an implicit leading one bit, and supports denormalized numbers.

Table 2 Rounding mode.

FPC byte 3 bits 6–7	*Rounding mode*
00	Round to nearest
01	Round toward 0
10	Round toward $+\infty$
11	Round toward $-\infty$

Table 3 Floating-point register layout.

128-bit FPR pair	
64-bit FPR	*64-bit FPR*
0	2
1	3
4	6
5	7
8	10
9	11
12	14
13	15

The bit representation of the BFP data formats in storage is defined to be left-to-right in a manner that is uniform for all numeric operands in the ESA/390 architecture. Although the format diagrams in the IEEE Floating-Point standard appear to use the same left-to-right bit sequence, the standard only defines the meaning of the bits without specifying how they appear in storage; the storage arrangement is left to the implementation. Several implementations in fact use other sequences; this may affect programs that depend on the bit representations of floating-point data in storage.

BFP short format

Figure 2 shows the bit distribution of a BFP short-format operand. When a number or NaN (Not-a-Number, a binary representation of a non-numeric character or string) in the BFP short format is loaded into an FPR, it occupies the left half of the register; the right half remains unchanged.

BFP long format

Figure 3 shows the bit distribution of a BFP long-format operand. When a number or NaN in the BFP long format is loaded into an FPR, it occupies the entire register.

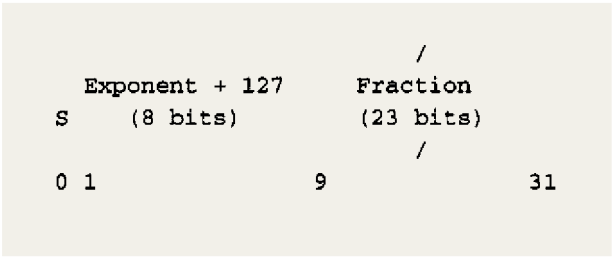

Figure 2

BFP short format (4 bytes).

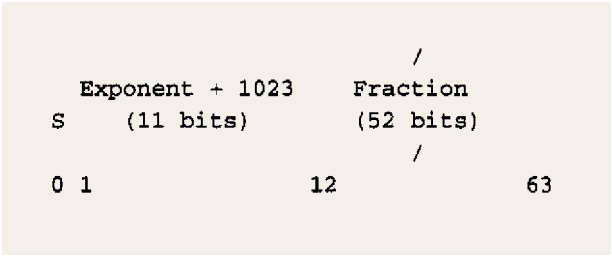

Figure 3

BFP long format (8 bytes).

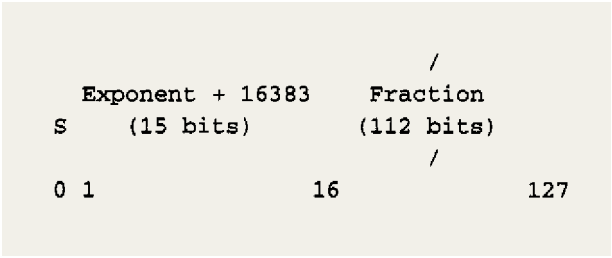

Figure 4

BFP extended format (16 bytes).

BFP extended format

Figure 4 shows the bit distribution of a BFP extended-format operand. A number or NaN in the BFP extended format occupies an FPR pair. The sign and biased exponent are in the leftmost 16 bits of the left register and are followed by the leftmost 48 bits of the fraction. The rightmost 64 bits of the fraction are in the right register of the pair.

The properties of the three formats are presented in **Table 4**.

NaN (Not-a-Number)

The standard requires two types of NaNs—signaling and quiet—where "Signaling NaNs afford values for [. . .] enhancements [. . .] not the subject of the standard" and "Quiet NaNs should [. . .] afford retrospective diagnostic information [. . .]" [6]. Quiet NaNs are normally propagated through arithmetic operations without signaling an exception. The standard does not specify the exact encoding of NaNs, but does indicate in an appendix that it is a mistake to use the sign bit to distinguish signaling from quiet NaNs. The encoding chosen indicates a quiet NaN by a one in the leftmost fraction bit. This encoding permits a signaling NaN to be converted to a quiet NaN by setting this bit to one.

When more than one NaN is encountered, the standard requires one of them to be propagated but does not specify which. Intel delivers the NaN with the larger fraction. RS/6000 delivers a NaN according to operand number. In ESA/390, signaling NaNs are chosen over Quiet NaNs, and NaNs of the same type are selected by operand number. This is similar to the technique used by the RS/6000.

- *Rounding*

The standard requires four rounding modes, with "round to nearest even" being the default. While most compilers do not specify the details for rounding for most operations, they do have special requirements in certain cases. For example, FLOAT, FIX, AMOD, AINT, and ANINT are normally expanded into in-line code that is sensitive to the details of rounding. To facilitate such operations, a special rounding-method field was defined to permit the current rounding mode to be overridden. This field permits the specification of all four rounding modes defined by IEEE, as well as "biased round to nearest," which is used by ANINT. The rounding-method field is provided for the following BFP instructions: CONVERT TO FIXED, DIVIDE TO INTEGER, and LOAD FP INTEGER.

- *Operations*

Considerations related to the way in which operations are performed were important in several respects.

Separate operation codes for each format

Several implementations of IEEE (Intel x86 and RS/6000, in particular) are optimized for the largest length supported. These machines convert shorter formats to a common format in a register, provide most operations only on the common format, and round (when required) to the precision of the shorter format. For ESA/390,

Table 4 Summary of BFP data formats.

Property	*Format*		
	Short	*Long*	*Extended*
Format length (bits)	32	64	128
Biased-exponent length (bits)	8	11	15
Fraction length (bits)	23	52	112
Precision (p)	24	53	113
Maximum exponent (E_{max})	127	1023	16383
Minimum exponent (E_{min})	−126	−1022	−16382
Exponent bias	127	1023	16383
N_{max}	$(1–2^{-24}) \times 2^{128}$ $\approx 3.4 \times 10^{38}$	$(1–2^{-53}) \times 2^{1024}$ $\approx 1.8 \times 10^{308}$	$(1–2^{-113}) \times 2^{16384}$ $\approx 1.2 \times 10^{4932}$
N_{min}	1.0×2^{-126} $\approx 1.2 \times 10^{-38}$	1.0×2^{-1022} $\approx 2.2 \times 10^{-308}$	1.0×2^{-16382} $\approx 3.4 \times 10^{-4932}$
D_{min}	1.0×2^{-149} $\approx 1.4 \times 10^{-45}$	1.0×2^{-1074} $\approx 4.9 \times 10^{-324}$	1.0×2^{-16494} $\approx 6.5 \times 10^{-4966}$

Explanation:
≈ Value is approximate.
D_{min} Smallest (in magnitude) representable denormalized number.
N_{max} Largest (in magnitude) representable number.
N_{min} Smallest (in magnitude) representable normalized number.

separate operation codes were defined for all three formats to match the implementation of HFP. This also provides maximum performance for all formats.

Storage-to-register operations
Most implementations of IEEE are optimized for register-to-register operations. The dataflow for S/390 processors is optimized for storage-to-register operations (inherited from System/360). To provide comparable performance for BFP and HFP, both register-to-register and storage-to-register operations were included. Storage-to-register versions of LOAD LENGTHENED provide loading and conversion as a single instruction. This also facilitates comparison between different formats, which is required by the standard. Storage-to-register operations for extended precision require additional data paths in the hardware and have not been implemented for either HFP or BFP.

Comparison
The standard permits comparison to be implemented either as a condition code or as a true–false response to a predicate. ESA/390 supports a condition code, whereas most compilers support predicates. Two instructions, COMPARE and COMPARE AND SIGNAL, were included to permit a compiler to support the predicate form.

Remainder
The remainder operation, required by the standard, is always exact. This operation may require thousands of cycles. The instruction DIVIDE TO INTEGER can be used to obtain the IEEE remainder by means of a simple two-instruction loop whose execution can be interrupted and restarted transparently.

Multiply–add
The instructions MULTIPLY AND ADD and MULTIPLY AND SUBTRACT are not required by the standard. They have the mathematical property of only one rounding. The definition is bit-for-bit compatible with the RS/6000, facilitating reuse of that system's library packages.

List of instructions
Table 5 is a list of instructions supported by the architecture for BFP operations.

- *Nontrapped underflow exceptions*

The standard defines underflow in terms of two events: tininess[2] and loss of accuracy. It permits each of these

[2] The term "tininess" refers to a nonzero number whose absolute value is less than E_{min}, where E_{min} is the smallest representable exponent (see Table 4). All denormalized numbers fall in this range.

Table 5 Instructions operating on BFP data.

Instruction name	*Source (bits)*	*32*			*64*			*128*		
	Result (bits)	*32*	*64*	*128*	*32*	*64*	*128*	*32*	*64*	*128*
ADD		R, S				R, S				R
COMPARE[a]		R, S				R, S				R
COMPARE AND SIGNAL[a]		R, S				R, S				R
CONVERT BFP TO HFP			R			R				
CONVERT HFP TO BFP*					R	R				
CONVERT FROM FIXED[b]		R	R	R						
CONVERT TO FIXED*[c]		R			R			R		
DIVIDE		R, S				R, S				R
DIVIDE TO INTEGER*[d]		R				R				
LOAD		R, S				R, S				R
LOAD AND TEST		R				R				R
LOAD COMPLEMENT		R				R				R
LOAD FP INTEGER*		R				R				R
LOAD LENGTHENED			R, S	R, S			R, S			
LOAD NEGATIVE		R				R				R
LOAD POSITIVE		R				R				R
LOAD ROUNDED					R			R	R	
LOAD ZERO[e]		R				R				R
MULTIPLY		R, S	R, S			R, S	R, S			R
MULTIPLY AND ADD		R, S				R, S				
MULTIPLY AND SUBTRACT		R, S				R, S				
SQUARE ROOT		R, S				R, S				R
SUBTRACT		R, S				R, S				R
STORE		S				S				
TEST DATA CLASS[a]		R				R				R

Explanation:
* Instruction includes an M field to control the rounding method.
[a] COMPARE, COMPARE AND SIGNAL, and TEST DATA CLASS set the condition code and do not produce a floating-point result.
[b] The source operand for CONVERT FROM FIXED comes from a general register.
[c] The result of CONVERT TO FIXED is placed in a general register.
[d] DIVIDE TO INTEGER provides the IEEE remainder function.
[e] LOAD ZERO does not have a source operand.
R Operation is provided in the register-to-register form.
S Operation is provided in the storage-to-register form.

events to be detected in either of two ways, but requires that the implementation detect them in the same way for all operations. Tininess may be detected either 1) after rounding[3] or 2) before rounding. Loss of accuracy may be detected as either 3) a denormalization loss or 4) an inexact result. Methods 2 and 4 were selected as being simpler and mathematically cleaner. This choice matches the RS/6000 implementation.

- *Traps*

Dependency on the control program is minimized by extending an existing program interruption, the data exception. The new conditions are identified by a data exception code (DXC) placed in the FPC register. The application program can access this information directly without requiring a control program service.

Trapped overflow and underflow on LOAD ROUNDED
The standard defines an exponent bias adjustment to be applied to the result for trapped overflows and underflows for "all operations except conversions." For conversions, the result must be returned in the same format or a wider format than the source. Exponent adjustment is permitted, but not required, for this case. In terms of the standard, the instruction LOAD ROUNDED is a "conversion."

For LOAD ROUNDED, overflow or underflow may occur in two ways: 1) in the original format, as the result of rounding to the precision of a shorter format, or 2) if the exponent of the result lies in the range of the source but not the target. In the first case, the exponent must be adjusted to fit within the range permitted by the source format. In the second, application of the bias adjustment value defined by the standard may cause the result to lie outside the range permitted by the format. The ESA/390 solution was to define a different bias adjustment value for LOAD ROUNDED that can be consistently applied in both cases.

[3] It should be noted that the phrase "after rounding" as used in the standard means "after intermediate rounding, but before final rounding."

Trap result incremented

The standard requires an implementation that always traps to indicate whether the delivered result has been rounded up. Although the ESA/390 implementation does not always trap, the data exception code (DXC) includes separate indications for "inexact and truncated" and "inexact and incremented." This information was included to assist the trap handler.

Conversion between decimal and binary floating-point

- *General statement of the problem*

Machine numbers are in binary; input to and output from programs is usually expected in decimal. Binary-to-decimal conversion is used for output of floating-point results; decimal-to-binary conversion is used for user-supplied input and for compile-time constants.

The IEEE standard on binary-to-decimal conversion strikes a compromise between what might be done efficiently in hardware (in early 1980s technology) and mathematical utility. For exponents in the center of the format's range (i.e., numbers whose absolute value is not far less than or greater than one), and for a decimal precision not exceeding the format's binary precision, a correctly rounded result is required; for numbers nearer the underflow or overflow thresholds, a bit more leeway is permitted. The standard explicitly requires that compile-time and run-time conversions of the same input yield the same result.

Floating-point decimal-to-binary conversion is usually provided by software, and the S/390 solution is no exception. Nevertheless, we wanted the conversion functions to carry the same expectation of exact reproducibility as one expects of a hardware instruction.

One way to achieve exact reproducibility in software is to prescribe a specific implementation in great detail. A better way is to prescribe a mathematically correct definition of the expected result; this gives the implementer the freedom to make tradeoffs suitable to a particular situation or, more significantly, to take advantage in the future of algorithmic improvements in terms of speed or resource consumption. (This is how the basic IEEE arithmetic operations are specified in the standard.)

For floating-point conversion, the mathematically precise definition is simply this: Perform the conversion as if infinite precision were available, then apply a single rounding step according to the intended rounding mode. In the case of the normal round-to-nearest mode, this means the result must be the single nearest representable result, or (in the case of an exact midpoint between two representable values) the nearest one whose low-order bit is zero.

The following observation reduces the problem to a finite-precision statement: Every binary (or hexadecimal) floating-point number has an exact decimal representation, because all factors of the input base (2) are also factors of the output base (10). Since the range of machine numbers is bounded, there is a longest decimal representation, and this bounds the required precision *if decimal arithmetic is used*. For example, the largest possible number of significant digits in the exact decimal representation of an IEEE Double is 751. The bound is one more than this, since we need to be able to distinguish the exact midpoint between two representable values. This number is $1/2\ D_{\min}$, or half the smallest denormal ($\approx 2.47 \times 10^{-324}$):

$$2^{-(1023+52)} = 0.000000 \cdots 0000002470328 \cdots 6328125$$

323 zeros and 752 digits

It may be interesting to illustrate how easy it is to do exact conversion using arbitrary-precision decimal arithmetic. The Rexx program shown in **Figure 5** displays the exact decimal value of an IEEE Double given in hexadecimal.

- *History*

Gordon Slishman gives a nice account [11] of how floating-point conversion accuracy has improved over the years, from quite poor in the 1960s to mostly acceptable in the late 1970s (in part due to the availability of extended precision for intermediate calculations). Within IBM, Assembler H and FORTRAN have had the highest standards, although an inconsistency between equally precise but different roundings in FORTRAN led Slishman to develop and publish correctly rounding conversions for HFP in the late 1980s. David Matula had published the theoretical underpinnings (e.g., [12]) and Jerry Coonen's analysis [13] led to the correctness bounds allowed by the IEEE standard. Unpublished work at IBM includes a decimal-arithmetic-based conversion routine for an IBM internal debugger (1977; the routines were also used in VM/XA in the mid-1980s). Unpublished work at MIT in the 1970s includes routines in MacLisp and ZetaLisp; this was disclosed in two seminal papers at the 1990 SIGPLAN conference [14, 15]. The work described here started in 1992 and evolved from an attempt to obtain provably correct routines for the debugger mentioned above, stimulated by the discovery of a few incorrectly rounded numbers (the search for which was in turn triggered by the discovery of a similar number for Assembler H).

Most programming languages or run-time environments that make any claim of accuracy imply no more than a (reasonably) bounded error. Rounding carry-propagation implies that several low-order binary (or decimal) digits may be wrong if they are all 0 or 1 (or 9). The Java

```
/* Convert IEEE Double-precision floater (given in hex) to decimal. */

numeric digits 800;       /* Sufficient for exact Double conversion */

arg hexfloat; hexfloat = left(space(hexfloat,0)||copies(0,16),16);

bex = x2d(left(hexfloat,3));     /* Biased exponent (plus sign bit) */
if bex > 2047 then do;
                   sign = '-';
                   bex = bex-2048;           /* Remove the sign bit */
                   end;
              else sign = '+';

if bex = 2047 then do;
  say 'Infinity or Nan: ' hexfloat
  exit;
end;

 exp = bex - 1023 - 52;
 if bex=0 then imant = 2 * x2d(right(hexfloat,13));    /* Denormal */
          else imant = x2d('1'right(hexfloat,13));     /* Normal   */

 decfloat = imant * 2**exp;

 /* An artifact of Rexx arithmetic may leave extra trailing zeros  */
 if index(decfloat,'.') > 0 then decfloat = strip(decfloat,tail,0);
 decfloat = format(decfloat,1,,3,0); /* Standard scientific layout */

 say 'Hex:'        hexfloat;
 say 'Dec:'  sign||decfloat;
 exit;
```

Figure 5

Rexx program showing exact decimal value of an IEEE Double given in hexadecimal.

language may be the first to require correctly rounded fixed-precision conversions. (Languages that support arbitrary-precision arithmetic present a different situation.)

- *Conversion basics*

Floating-point numbers are traditionally expressed as

$$fraction \times base^{exponent}.$$

An equivalent representation that exposes the precision (expressed as a number of bits or digits) is

$$(integer + fraction) \times base^{exponent - precision}.$$

When the precision of the input to conversion is implicit in the number of bits or digits given, the fraction (in the exposed-precision representation shown above) will be zero. The fraction may be nonzero when this representation is used to show the last intermediate result of conversion, just before the rounding step. The rounding rule takes the fraction into account when determining whether the integer in the final result should be one more than the integer in the intermediate result.

The basic method for converting a floating-point number from one base to another is through a sequence

of transformations that leave the value intact. The starting point is

$value = infract \times inbase^{inexp}$ (inbase = 2 or 10).

The ending point should be (for the same value)

$value = outfract \times outbase^{outexp}$ (outbase = 10 or 2).

In principle, the internal representation used during the conversion process can differ from both input and output representations—it might be fixed-point binary, for example, when converting between decimal character strings and machine floating-point formats. This leads to the following three-step overall description of conversion:

1. Input conversion to internal format.
2. Value-preserving transformation.
3. Output conversion from internal format to final result.

The internal format used for input conversion is chosen to make that step as straightforward as possible, and the same argument applies to the internal format expected by the output conversion. These internal representations are not the same; hence the need for a value-preserving transformation.

We know how to convert integers from one base to another. There are machine instructions to convert integers that fit in a register, and techniques for converting larger integers using multiple-precision fixed-point arithmetic are well known. The actual conversion between decimal and binary (whether input, output, or intermediate) is easiest when the value is in the exposed-precision representation, where the numbers to be converted are integers.

• *Using higher-precision floating-point arithmetic*
Mathematically, the transformation from the internal input representation to the internal output representation can be written very simply:

$outfract = infract \times factor$,

where

$$factor = \frac{inbase^{inexp}}{outbase^{outexp}}.$$

The output exponent is chosen to make this factor close to one. The number of possible exponent values is limited by the format, so this computation is typically carried out by a combination of table lookup and a small number of floating-point multiplications. If the computation can be carried out at a higher precision than the precision of the desired result, the error due to repeated rounding can be kept reasonably small—but the final result will in general not be the same as if infinite precision had been used (in the constants as well as in the arithmetic), followed by a single rounding.

If the constants are carefully chosen and the rounding errors are carefully tracked, it is possible but nontrivial to determine when the conversion result is in fact the correctly rounded result; this is the case when the rounding error is too small to bridge the gap to the next representable number. Some correctly rounding conversion methods do this, and retry the conversion of *difficult* numbers with higher precision—typically using multiple-precision arithmetic. (A *difficult* number is one whose value is uncomfortably close to a rounding threshold—see the subsection below on difficult numbers.)

Note that when the precision of the available machine is no greater than the precision of the desired result, multiple-precision arithmetic may be needed even in the common case. The S/390 conversion routines apply the same standard of correctness to 128-bit extended-precision conversion, so we have to face this issue.

• *Using multiple-precision integer arithmetic*
The conversion routines provided with the S/390 IEEE Floating-Point support do not actually use floating-point arithmetic at all; they treat the floating-point formats as bit patterns to be processed, and use multiple-precision integer arithmetic. This has several advantages:

- The core conversion routines can be independent of the input and output formats; only generic parameters such as precision and exponent range are required. Extended-precision conversion is no more difficult than single-precision; it simply requires more resources. (The High-Level Assembler also takes advantage of generic conversion by offering improved conversion for hexadecimal floating-point together with support for IEEE binary floating-point.)
- The handling of difficult numbers is not essentially different from the easy case. This improves the robustness of the implementation.
- Proving correctness is conceptually easier than for floating-point arithmetic. This improves confidence in the implementation.
- The conversion routines can run efficiently on hardware that does not yet support the new floating-point instructions. This is especially useful for compilers and assemblers.

The internal representation is as a "bignum" fixed-point number multiplied by a power of two as well as a power of ten:

$(integer + fraction) \times 2^{binexp} \times 10^{decexp}$.

By *bignum* we mean a number expressed in a large base, such as base 10^9 or 2^{32}, where each digit can be processed

as a 32-bit machine integer. A decimal bignum uses a power of ten as a base; 10^9 is a good choice because it nearly fills a 32-bit register. This representation allows for simple and efficient multiple-precision arithmetic. A binary bignum uses a power of two as a base. Conversion to and from bignums of a compatible base is easy, because it can be done simply by grouping smallnum digits or bits (as the case may be).

A fixed-point bignum consists of an array of 32-bit words in storage; the position of the radix point is implicit (i.e., the software knows where it is), and is chosen to reflect the desired output precision. It should be large enough to accommodate both the given input precision and the desired output precision. In particular, the rounding position should be in the integer part, so that the only knowledge required of the fraction is whether it is exactly zero or not. The actual fraction will then not have to participate in output conversion.

The input and output conversions convert to or from a compatible bignum base. Let us describe decimal-to-binary conversion, to be definite, and revisit the three essential conversion steps mentioned earlier:

1. Input conversion from a decimal character string to a decimal bignum times a power of ten.
2. Transformation of the decimal bignum times a power of ten into a binary bignum times a power of two.
3. Output conversion from a binary bignum times a power of two to the desired binary machine format.

Input conversion is straightforward. The only point worth mentioning is that an indefinite number of digits is accepted, but if more digits are given than are necessary to represent the longest exact decimal representation, the extra digits are only checked to see whether they are all zero; in effect, they are coalesced into a single "sticky" digit. This is necessary in order to be able to break rounding ties correctly, and useful in that it allows the internal storage requirements to be bounded by the result format, without restricting the input format.

Output conversion turns out to be the trickiest part, even though it is not conceptually demanding. This is where the single rounding step takes place, and one needs to be careful to account for all of the bits that are (or might have been) beyond the rounding point, in order to break ties correctly in all cases. The rounding position has to be determined relative to the high-order bit, and has to take possible denormalization into account.

• *A core conversion algorithm*

The core of the conversion routine is the transformation of the internal representation. We illustrate here the algorithm for decimal-to-binary conversion. Binary-to-decimal conversion is essentially the same, but in the opposite direction.

The algorithm transforms the internal representation without changing the represented value. At all points, the representation is as follows, with *value* being invariant:

$$value = (integer + fraction) \times 2^{binexp} \times 5^{decexp}.$$

At the beginning, we have a decimal bignum with a null fraction, and $binexp = decexp$ (because we are given a power of ten, not a power of five).

At the end, we want *decexp* to be zero, with a binary bignum. The size (number of bigdigits) of the integer part of the binary bignum must be such that the rounding point will be inside the integer part, with at least one bit to the right of the rounding point. It is then sufficient to know whether the fraction is exactly zero or not in order to perform correct rounding, including correct tie-breaking.

This transformation is called *exponent reduction*. It is achieved by multiplying the bignum by a sequence of factors of the form $5^a/2^b$, where a and b have the same sign as *decexp*. Each factor should be close to one, so as not to change the size of the integral part of the bignum. Indeed, the algorithm switches dynamically between a factor that is smaller than one and a factor that is greater than one, in order to keep the high-order bigdigit within a narrow range. The cleverness of the algorithm lies in the careful choice of the exponents a and b.

When the initial *decexp* is positive, we would be multiplying by a power of five and dividing by a power of two. This suggests that a binary bignum is most appropriate, because if we choose 2^b to equal the binary bigbase (i.e., $b = 32$), the division is free, because it amounts to no more than occupying a new fraction bigdigit, since the result of the multiplication is implicitly shifted one bigdigit to the right. Since the factor should be close to one, this choice of b suggests $a = 14$ for a factor of $5^{14}/2^{32} \approx 1.4 > 1$, or $a = 13$ for a factor of $5^{13}/2^{32} \approx 0.28 < 1$.

When the initial *decexp* is negative, we would be multiplying by a power of two and dividing by a power of five. This suggests that a decimal bignum is most appropriate, because if we pick 10^a to equal the decimal bigbase (i.e., $a = 9$), the division comes cost-free again. The two factors used to achieve a balanced negative exponent reduction are $2^{30}/10^9 \approx 1.05 > 1$ and $2^{29}/10^9 \approx 0.54 < 1$.

At some point, the bignum must be converted from bigdecimal to bigbinary. It is now clear that this conversion should be done before the multiplicative transformation when the exponent is positive, and afterward when it is negative. In other words, we use either bigbinary or bigdecimal arithmetic, depending on whether we are converting large or small magnitudes. Performing the conversion as the first or last step also

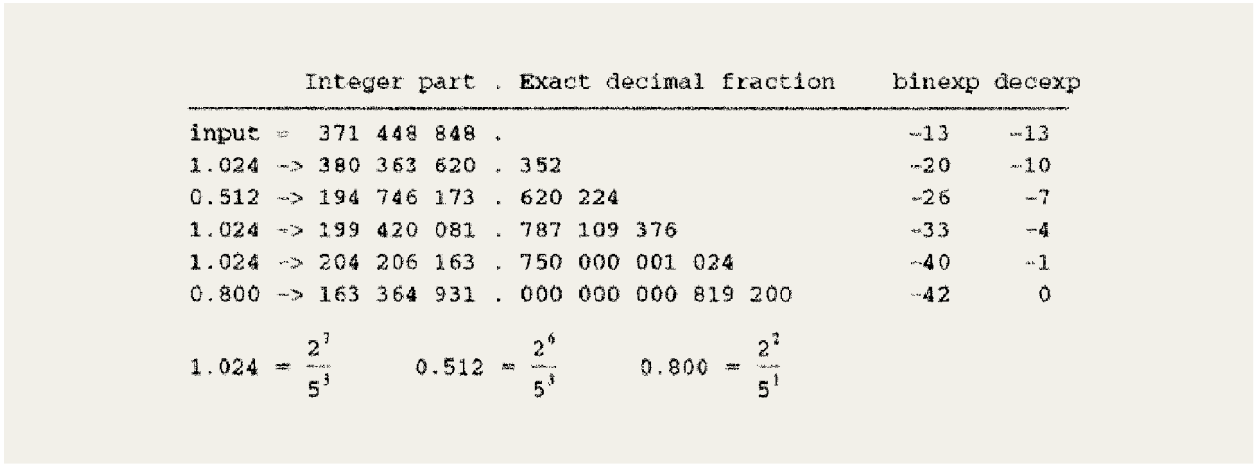

Figure 6

Example determination of required bignum size.

avoids the need to convert the fraction; it is zero initially, and at the end its actual value does not matter—only whether it is zero or not.

Multiplication by the factors considered here is much simpler than general bignum multiplication; it almost degenerates to the case of multiplying one bignum by a single-digit bignum. The factor that is smaller than one is clearly representable by a single bigdigit. The factor that is larger than one does not exceed twice the bignum base, and it turns out that this is only marginally more difficult than single-bigdigit multiplication when the bigbase is 2^{32}, and comes cost-free when the bigbase is 10^9 if we allow the high-order bigdigit to exceed the bigbase (this works because decimal bigdigits are stored in a binary box of size 2^{32}, and $2 \times 10^9 < 2^{31}$).

Exponent reduction by means of the factors described above reduces the absolute value of the decimal exponent by 9, 13, or 14 per step, so the last step (to reduce the remaining decimal exponent to zero) may have to use a different factor, chosen from a small table of residual factors of the same general form. For example, to reduce a residual decimal exponent of +5, use the factor $5^5/2^{12}$, which is $(2^{15} \times 10^5)/2^{32}$, and to reduce a residual decimal exponent of −5, use $2^{12}/5^5$, or $(2^{17} \times 10^4)/10^9$.

Note that, because of the choice of decimal arithmetic for reducing negative exponents, all calculations described above are *exact* if we allow the bignum fraction to grow by up to one bigdigit per multiplication. The maximum number of multiplications is determined by the valid exponent range, and is therefore bounded by the format of the binary floating-point number (whether input or output). There are no difficult numbers for this method. The long fraction resulting from the conversion of extreme exponents will be discarded after checking whether it is zero or not, because that is all that is needed to figure out how to round the integral part correctly.

The determination of the required bignum size (which controls the resource requirements of the algorithm) must take into account several factors. The integral portion must accommodate the result precision, even when the initial high-order bigdigit is only 1/10th of the base and when the cumulative factor applied during exponent reduction is as small as it can be. The self-balancing nature of exponent reduction makes the cumulative factor come close to one, but one must know a lower bound. The fraction must accommodate the largest possible number of reduction steps (each can contribute at most one new fraction bigdigit), and this is determined by the format-derived exponent bound.

Figure 6 shows a worked-out example. We use a decimal bigbase of 1000 in order to illustrate the procedure. Each group of three digits represents one bigdigit. The reduction factors would be 1.024 and 0.512 in this case. The given input is 3.71448848e−5 (or $371448848 \times 10^{-13}$), to be converted to IEEE Single (with a 24-bit fraction). The result is 163364931×2^{-42} plus a very small increment whose only significance is that there will be no tie-breaking during rounding after 24 bits, which is indeed safely within the 28-bit hexadecimal number 9BCC043. Normalizing to one bit before the binary point yields $1.00110111100110000000100_2 \times 2^{-15}$ for a biased exponent of 127 − 15 = 112 (hex 70). Putting the pieces together yields the hex representation of 381BCC04 for this IEEE Single.

```
        Integer part . Exact decimal fraction
-------------------------------------------------------
input =  371 448 848 .
1.024 -> 380 363 620 . 352
0.512 -> 194 746 173 . 620 224
1.024 -> 199 420 081 . 787 109 376
1.024 -> 204 206 163 . 750 000 001 024
0.800 -> 163 364 931 . 000 000 000 819 200

        Integer part . Truncated fraction  (two fraction bigdigits)
-------------------------------------------------------------------
input =  371 448 848 .
1.024 -> 380 363 620 . 352                 (same int) . 352
0.512 -> 194 746 173 . 620 . . .           (same int) . 620 224
1.024 -> 199 420 081 . 786 . . .           (same int) . 787 109 . . .
1.024 -> 204 206 163 . 748 . . .           (same int) . 749 999 . . .
0.800 -> 163 364 930 . 998 . . .           (same int) . 999 999 . . .
```

Figure 7

Example exponent reduction with truncation after one and after two bigdigits.

• *Improving performance using truncated bignum multiplication*

The exact fraction can grow to be quite large, especially for extended precision—and yet it will be discarded at the end. It would seem certain that a better approach is possible.

Suppose we keep only one fraction bigdigit (i.e., we discard all new fraction bigdigits after the first). (We do need to keep track of the zeroness of the fraction.) The result will certainly not be greater than the exact result, but it may be smaller. Each truncated multiplication drops data beyond the last surviving fraction bigdigit, so the incremental error is bounded by one in the value of this bigdigit. Each error is multiplied by the cumulative reduction factors that follow, but those are close to one, and the largest relevant factor is bounded independently of the number of multiplications. The cumulative error is therefore proportional to the number of multiplications. This number can be determined from the exponent to be reduced, so we have a useful bound on the total truncation error. With a bigbase of 10^9, this error is at most a few thousand across the entire exponent range of BFP extended precision. A blind error threshold of 5000 would be sufficient (in fact, the program picks the error threshold dynamically). With this, any first-fraction bigdigit exceeding 999995000 would be suspicious; with "random" inputs, this might happen once out of 100 000 conversions.

When the truncated result is suspicious, we can retry the exponent reduction with higher precision, but that might still not be enough, as revealed by the example shown in **Figure 7**. The cost of exact multiplication is not grossly worse than that of truncated multiplication—factors of 2 to 10 in execution time were measured (for double precision). Since the actual retry probability (using tight error bounds, and two fraction bigdigits when the number of multiplications is large) is less than 1 in 10^7, any retry is in fact done exactly.

The conversion example shown earlier can be used to illustrate the effect. We repeat the exponent reduction in Figure 7 with truncation after one and after two bigdigits (base 1000 in this example). The error threshold in this simple example would have been 5, and this last fraction bigdigit exceeding 995 would have been judged "suspicious," for a good reason: The exact answer is indeed different before the decimal point.

The example shown in **Figure 8** illustrates that a truncated result may look *very* suspicious and yet be correct. It shows a different kind of difficult number. The exact and truncated values are shown side by side here. The number is 5.29097127e−7.

```
        Integer part . Exact fraction      With truncated fraction
-------------------------------------------------------------------
input =  529 097 127 .                     (same int) .
1.024 -> 541 795 458 . 048                 (same int) . 048
0.512 -> 277 399 274 . 520 576             (same int) . 520 576
1.024 -> 284 056 857 . 109 069 824         (same int) . 109 069 . . .
1.024 -> 290 874 221 . 679 687 499 776     (same int) . 679 686 . . .
1.024 -> 297 855 202 . 999 999 999 770 624 (same int) . 999 998 . . .
```

Figure 8

Example exponent reduction illustrating a correct but suspicious-looking truncated result.

• *Difficult numbers*

A difficult number is one whose binary (decimal) representation has a very long run of zeros or ones (nines) near the rounding point. Such a long run permits tiny errors to affect the bits that are to be retained, because of carry propagation.

In the context of our conversion algorithm, they are numbers that cause the initial (integral) bignum *bigbeg* to be transformed to a near-integer result *bigend* by multiplying it by a factor of the form $2^x/5^y$ whose value is close to 1 (in the range 1/5 to 5; x and y are integers of the same sign):

$$bigend = bigbeg \times \left(\frac{2^x}{5^y}\right) \pm tinyfraction.$$

If $tinyfraction = 0$, then one of *bigbeg* or *bigend* is a power of 2 and the other is a power of 5.

The continued-fraction expansion of a real number leads to a sequence of progressively better "best" rational approximations of the number, where "best" means that there is no better rational approximation with a smaller (reduced) denominator. Continued-fraction expansion of a rational number such as $2^x/5^y$ leads to a finite sequence of rational approximations, the last of which is exact. The point is that the given numerator and denominator may be very large, whereas the partial convergents start out as very simple fractions (the crudest being 1/1 when the rational number is between 1/2 and 2), and require progressively more digits as the approximations converge to the true value.

Our difficult numbers correspond to partial convergents of such fractions. The examples used above were found by looking for nine-digit partial convergents for increasing exponents x in $2^x/5^y$ (with y chosen to keep the ratio close to one). One can find thousands of examples of this kind for single and double precision, and hundreds of thousands for extended precision, dispelling any hope of simply checking for a handful of exceptions. (For large exponents, as are encountered with extended precision, each continued-fraction expansion may yield several thousand partial convergents, and for each power of ten there are four or five ratios in the range (1/5, 5) to try, since $2^4 < 5^2 < 2^5$. Not every convergent leads to a good difficult number (because the run of zeros or nines may be too short, or may not occur near enough to the rounding point), but a significant proportion of them does.

The length of a run of zeros or nines (ones) can be shown to be bounded by the size (in digits or bits) of the initial bignum plus the size of the partial quotients at the point where the partial convergent is taken. This means that if we could bound the size of the partial quotients that occur in the continued-fraction expansions of near-unity ratios of the form $2^x/5^y$, we could bound the bignum fraction length required to guarantee the correctness of the integral part without having to resort to exact multiplication.

The number of possibly relevant expansions is bounded by the exponent range of the format being converted, so a bound on partial quotient size can be obtained by exhaustive search. This is time-consuming and not very elegant, but preliminary experiments suggest that this bound may be surprisingly small: The largest partial quotient encountered across the range of possible exponents had nine digits (28 bits). This is small, considering that the corresponding convergents contain up to 21000 digits. (For the range of exponents possible with double precision, the largest partial quotient was the seven-digit number 7651576 (23 bits).

A future implementation of the conversion algorithm may be able to take advantage of this observation, if

confirmed or rigorously proved. It would be especially advantageous (in terms of the required resources) in cases in which the size of the decimal input or output is restricted to a number of digits commensurate with the binary precision (9, 17, or 35 digits), because the bignum fraction would have to be only one or two bigdigits longer than the integral part (which in turn would require at most four bigdigits).

Thanks to a single Assembler H rounding error discovered in the late 1980s, we are now able to generate many more difficult numbers. Here are some examples:

```
DC E'.10537713134640190603190040568O4E-41'
```
31-digit HFP single

```
DC D'.303325544866797714604E-10'
```
21-digit HFP double

```
DC L'.8031692147E-10'
```
10-digit HFP extended

High-Level Assembler Release 3 (see the subsection on the High-Level Assembler) supports the new conversion routines in addition to the old ones, so the reader might like to compare results for these difficult numbers. In particular, comparing types `D`, `DH`, and `L` for the HFP double example, or types `E`, `EH`, and `D` for the HFP single example, demonstrates that the old types (`E` and `D`) fail to round up. The single-precision number is particularly stunning, because it looks like an exact tie even in extended precision; it takes 15 bits more than 128-bit extended precision to show that there are additional nonzero bits, and that the number should be rounded up even if IEEE rounding were applied, where an exact tie should have rounded down.

- *Conversion resource requirements*

The resources required for conversion are time and scratch storage. The primary factors affecting storage requirements are the covered exponent range and the number of decimal digits allowed. The decimal size may be unbounded, because there is a natural bound derived from the exponent range, as was mentioned earlier. This bound is 126 digits for single, 752 for double, and 11503 for extended precision.

The most expensive single operation (in terms of execution time) is the decimal-to-binary conversion of the bignum integer; it is quadratic in the number of bigdigits. This is an issue only when thousands of digits (leading to hundreds of bigdigits) are given for extended precision (because of the implied maximum for double and single precision). In practice, only as many digits as make sense for the desired precision are given (some languages do not even allow more), in which case there are at most four integral bigdigits, and this is not an issue.

Exponents near the extreme of the range require many exponent reduction steps (bignum multiplications where one factor has a single bigdigit). This number is roughly one-tenth of the absolute value of the decimal exponent. For difficult numbers, the exact retry can be expensive (because of the growing fraction), but one would have to compile a large table where *all* numbers are difficult for this to be a problem.

The storage requirements for the maximum possible fraction growth must be met, however, because no matter how rare the need may be, a failure cannot be tolerated. In the case of decimal-to-binary conversion, negative exponents are doubly expensive: Exponent reduction reduces the remaining decimal exponent by only 9 per step (instead of 13 or 14 for positive exponent reduction), and the largest starting exponent (in absolute value) can be very large because it must account for the input precision (the given number of digits) as well as for possible denormals. The worst case is extended-precision $D_{\min}$ (roughly 4e−4966) given as an 11503-digit exact decimal. The starting decimal exponent in that case would be −16470, requiring up to 1830 fraction bigdigits! (This is in addition to the 1279 bigdigits required to hold the integer part of this bignum.)

The work area size (in bytes) turns out to be $256 + 0.45E_{\max} + 1.34D_{\max}$, where $E_{\max}$ is the highest decimal fraction exponent (38, 307, or 4915) and $D_{\max}$ is the given or implied bound on the number of decimal digits. This is at most 18 KB (extended precision, unbounded decimal length). It is 2.5 KB when restricted to 36 digits and only a few hundred bytes when further restricted to double. For double with unbounded decimal length (e.g., as required for the Java language) it is about 1.5 KB. When constraints are known in advance, a fixed work area can be used; alternatively, when the conversion fails because of insufficient room, one can expand the work area dynamically and then retry the conversion.

(If the bound on continued-fraction expansions is confirmed, the dependency on $E_{\max}$ would be removed from the space requirement, and a few hundred bytes would suffice even for extended precision if the decimal precision is limited to, say, 36 digits.)

- *Extra considerations for HLASM*

The Assembler has traditionally supported the generation of artificially shortened constants by means of an explicit length declaration (overriding the natural length of the format). This length could even be specified in bits, e.g., DL.12′1.2,3.4′ would assemble two 12-bit "double-precision" floating-point numbers in three bytes, with rounding in the last retained bit position.

The generic conversion algorithm described here is totally unconcerned about this, since it supports *any*

fraction length, and there is nothing special about the lengths implied by the standard formats.

The High-Level Assembler was thus able to retain this idiosyncrasy for binary floating-point formats (it is difficult to call DBL.27′123′ an "IEEE Double").

Software support

Support for IEEE Floating-Point involves both hardware and software. In this section, software-related aspects are described.

- *OS/390 Control Program support*

The OS/390 Base Control Program (BCP) and Base UNIX System Services were enhanced to support IEEE Floating-Point.

Base Control Program support

The OS/390 Base Control Program [27–29] support for IEEE Floating-Point underpins all of the higher layers of software. Since it is more convenient for the upper layers, especially applications, to base their initial use of IEEE Floating-Point on a given level of OS/390, rather than being restricted to S/390 processors that support the FP extensions, the BCP also simulates the new instructions and registers at the machine-code level.

Support of hardware

BCP support for the FP extensions consists of detecting the presence of the new hardware and enabling its use by software, including the new registers in status saving, handling new hardware exceptions, and extending RAS support to include the new registers.

Detection and enablement Although the FP extensions are always available on all of the CPUs of a G5 processor, it would be dangerous to allow an application to use these facilities on an older level of the BCP, since the AFPRs and FPC register would not be saved and restored on interrupts and task switches. Therefore, as described above in the subsection on activating additional registers, the BCP must enable use of these facilities by setting on the CPU's AFPR-control bit (CR 0.13). Besides providing compatibility, this requirement has the additional benefit of reducing supervisor overhead for saving and restoring the new registers; this overhead is incurred only for dispatchable units[4] that actually use them.

During IPL the BCP determines whether the FP extensions are available. Because multiple programs may wish to check for its presence, a general use programming interface (GUPI) is provided by a flag in the communications vector table (CVT). For programs that require the capabilities of the FP extensions but make only light use of them, a second CVT flag indicates whether floating-point simulation is available.

After IPL, if the FP extensions are available, the BCP selectively enables its use by application programs. Enablement occurs transparently to the dispatchable unit as follows:

1. When a program tries to execute a BFP instruction or use an AFPR with CR 0.13 off, the CPU presents a data exception with a special DXC value.
2. The program check handler in the BCP recognizes the special exception and enables CR 0.13 and supervisor status saving of the AFPRs and FPC register. For a task, a flag is set in the secondary task control block (STCB) to allow upper software layers to determine whether it is using the new registers.
3. The program check handler then redrives the instruction, and the program continues execution.

The IEEE 754 standard defines default settings for the rounding mode and exception mask. When the BCP enables a dispatchable unit to use the FP extensions, it establishes these defaults by clearing the FPC register to zero.[5]

Normally, once a dispatchable unit is enabled to use the FP extensions, it remains enabled for its lifetime, and the system incurs a small status-saving overhead even if the dispatchable unit stops using the new registers. While not a significant penalty for normal applications, a long-lived dispatchable unit might wish to disable use of the new registers and discontinue the associated status saving when these are no longer needed. For example, a transaction manager scheduling many transactions under the same task, where only a few transactions use FP extensions, might find this advantageous. A BCP service provides the disable function.

New register status saving The supervisor must include the AFPRs and FPC register when saving and restoring status for dispatchable units that are enabled to use them. This occurs when the dispatcher switches the current dispatchable unit and on some interrupts. Although floating-point is normally used by applications running in task mode, the new registers must also be included in service request block (SRB) status saving for subsystems such as DB2 that use SRBs to process application requests.

Another area where the BCP could choose to save and restore the AFPRs and FPC register is in system linkages, such as LINK and XCTL, and supervisor calls (SVCs).

[4] *Dispatchable unit* is an S/390 term for a unit of work that is performed through execution of a program by one CPU at a time, e.g., an OS/390 task or service request block.

[5] An application run-time environment must perform its own initialization of the FPC register, however, because some program could have run under the unit of work prior to run-time initialization and set the FPC register.

Similarly, the hardware could add the new registers to the status saved or restored by the PC/PR/PT and BAKR instructions. It was decided, however, to continue the previous policy of allowing the FPRs to be managed by the application within the scope of a dispatchable unit. Besides being compatible with the past, this approach

- Reduces status-saving overhead, since system code rarely uses the FPRs.
- Avoids development cost in the BCP to add floating-point status saving at multiple points.
- Avoids behavior and performance changes for the PC/PR/PT and BAKR instructions.
- Allows the application program full control of the FPRs.

The last point is also a drawback, however, since an application program must be aware that it is responsible for saving and restoring the new registers across these linkages, if needed.

Hardware exceptions The BCP has minimal involvement in handling IEEE Floating-Point exceptions. The extended synchronous program interrupt exit (ESPIE) interface allows an application (or run-time environment) to enable for certain HFP exceptions (in the program mask field of the PSW) and gain control in an exit routine when they occur. For BFP, all IEEE exceptions occur as data exceptions. The application manages the exception mask and flags in the FPC register. When an IEEE exception occurs, the program check first-level interrupt handler (FLIH) processes the data exception normally. The only additional support required is to copy the DXC value from low storage and pass it to the application program's ESPIE routine or, via the recovery termination manager (RTM), to the recovery routine for the dispatchable unit [functional recovery routine (FRR) or extended specify task abnormal exit (ESTAE)], if either of these is established. The ESPIE or recovery routine accesses the FPRs directly, if needed.

Reliability, availability, and serviceability (RAS) A critical area of system reliability and availability involves machine check handling and alternate CPU recovery. In these cases, an error in the CPU has occurred and its status may have to be saved for software recovery, including possibly moving the current dispatchable unit to another CPU. To enable such recovery, the G5 processor includes the AFPRs and FPC register as part of the CPU status. For serviceability, the AFPRs and FPC register are included and formatted in the logging of CPU errors. The new registers are saved when a standalone dump is taken in the event of a system failure. Application and system dumps and formatters also include the new registers.

Scheduling IEEE Floating-Point work in a sysplex Applications using the FP extensions and running in a parallel sysplex environment must be scheduled on a system with the hardware, or at least the software, that supports them. This requirement was anticipated in an earlier OS/390 release by the WorkLoad Manager (WLM) [16]. The WLM allows an installation to identify jobs that require specific hardware, software, or data resources and to define systems as having, or not having, those resources. The WLM then ensures that jobs are scheduled only on systems with the required resources. This capability can be used to direct applications using the FP extensions to systems in a sysplex running on a G5 processor or, for light use, to systems with floating-point simulation.

Floating-point instruction simulator The primary goal of the simulator is to allow application programs that exploit the FP extensions to run on OS/390 Version 2, Release 6 when a G5 processor is not available. With the exception of speed, a program will function identically. Secondary goals are to minimize the differences to ancillary functions such as tracing when using simulation and to make the simulation paths relatively efficient.

Implementation Simulation of the FP extensions entails not only performing floating-point operations and detecting associated exception cases, but also detecting most other conditions that can occur when a CPU executes an S/390 instruction (e.g., storage access and specification exceptions). To perform this kind of simulation, software must

1. Receive control on each instruction to be simulated.
2. Determine the instruction (operation code) to be simulated.
3. Decode the instruction, generate the result, and detect any exceptions that would occur in hardware execution.
4. Return to the point that would receive control in hardware execution (continue to the next sequential instruction or process an exception).

It is more efficient to perform this level of simulation very close to the hardware, so it is implemented as a second-level interrupt handler (SLIH) that receives control from the program check first-level interrupt handler (FLIH).

The FLIH receives control for instruction simulation when the FP extensions are not available and the CPU encounters either a new floating-point instruction (operation exception) or an existing floating-point instruction using an AFPR (specification exception). Normally these exceptions would be presented to the program's ESPIE or recovery routine as an error. In

OS/390 Version 2, Release 6, however, when not running on a G5 processor, the FLIH invokes the simulation SLIH. To avoid clogging the system trace table with unnecessary entries, program interruptions for simulated instructions are not traced.

The simulation SLIH fetches the instruction and determines whether the operation is one of those to be simulated. To detect storage access exceptions that the user's program would trigger if running on the actual hardware, the SLIH runs in the address-space control (ASC) mode of the user's program to fetch the instruction, and accesses user storage with the user's key.

The job of fetching the instruction is further complicated by the fact that the instruction typically resides in pageable storage, but the SLIH is running with the CPU disabled for interrupts. If a page fault occurs, it must be treated like a normal page fault. This is done by setting a footprint flag before the instruction fetch, so that if the FLIH is re-entered for a page fault, it is known that this happened on a simulated instruction. The FLIH resets the environment so that the page fault appears to have occurred on the original instruction and is handled normally.

A table-driven routine is used to decode the instruction and invoke the processing routine for it. The task of creating software routines to faithfully imitate the more than 100 new floating-point instructions in the G5 processor was made less daunting by reusing the algorithms from a hardware test tool. Using the same algorithms in the software simulation also helped to ensure that results were identical with those of the actual G5 hardware.

The algorithms simulate all functions of the hardware, including setting the condition code, checking and setting fields in the FPC register, and detecting exception conditions. The original FPRs (0, 2, 4, and 6) are used to improve performance of the simulation. Page faults that occur when fetching operands or storing a result are handled as described earlier for instruction fetch.

If no exceptions are detected by the simulation, the SLIH returns to the FLIH to continue with the next sequential instruction in the user's program. If an exception is detected, the SLIH directs the FLIH to change the original program interruption to the appropriate exception type and handle it as if it had occurred in the hardware.

Validation To ensure that it produces results identical to those of the G5 processor, the floating-point simulator underwent rigorous testing using the same test suite as the hardware test tool.

Performance and side effects Naturally, the CPU time used to simulate a floating-point instruction is vastly greater than with the G5 hardware. On a per-instruction basis, the simulator is at least 100 times slower.

Time spent in the simulator could be counted as CPU time for the user's program ("captured" time) or considered to be system overhead ("uncaptured" time). Although in all other ways the simulator behaves identically to the hardware, the need to account for the CPU resource consumed by user programs requires that simulator time be captured. As a result, performance reporting tools such as the Resource Measurement Facility (RMF) show higher CPU times for jobs run using the simulator.

Base UNIX System Services support
OS/390 includes standard UNIX functions that replicate, reuse, or create new environments [17–20]. Some of these functions must be changed to initialize or copy the AFPRs and FPC register. However, no action is taken if the task's STCB flag indicates that the new registers are not enabled for use.

- fork() creates a copy of a process. The AFPRs and FPC register are copied from the parent process to the child process.
- exec() creates a new execution environment with a new task. The FPC register is set to the IEEE 754 standard defaults, as discussed above in the subsection on detection and enablement.
- spawn() is logically a fork() followed by an exec(). The initialization is the same as for exec(). For a medium-weight spawn(), however, the task is reused, and the BCP service is called to disable use of the AFPRs and FPC register.
- pthread_create() produces a new heavy- or medium-weight thread, running as a new task. The FPC register is copied from the initial pthread-creating task (IPT). When a medium-weight thread exits and gets new work, the enablement state of the AFPRs and FPC register, and the FPC register itself, are reset to their original conditions for the thread.

OS/390 UNIX also includes the ptrace() function, which allows debuggers, such as the OS/390 UNIX System Services Debugger, to obtain information about and control another process. Support is added to ptrace() to set and get the AFPRs and FPC register for a thread.

Although not part of the base UNIX support, the *od* command in the UNIX Shell and Utilities deserves mention; *od* reads data from a file and formats it for output, according to the data type specified by the user. This command supports floating-point data types and was updated to allow the user to specify that data is in BFP or HFP format.

• *VM/ESA support*

VM/ESA supports FP extensions to enable guest virtual machines to exploit all aspects of the new architecture.

Host support

The VM control program itself makes only minimal use of the new facilities. In particular, it

1. Detects whether they are installed, in order to determine whether they can be provided to guests.
2. Creates extended save areas so that a machine check saves the AFPRs and FPC register for use in error recording and to preserve guest integrity.
3. Ensures that guest AFPRs and FPC register contents are saved and restored appropriately.

Guest operation

VM hosts support guests at multiple levels. A virtual machine operating directly on the host VM system is called a *real guest*, or *R-guest*. Real guests can run VM support guests of their own. A virtual machine operating on a real guest is called a *virtual guest*, or *V-guest*.

When an R-guest dispatches a V-guest using the interpretive execution facility, the host VM intercepts that dispatch and runs the V-guest directly, much as it runs an R-guest. Consequently, when a V-guest running VM/ESA dispatches a virtual machine (conceptually a *V-V-guest*), the dispatch is intercepted by the R-guest, which tries to run the V-V-guest directly, as a V-guest of the host. That dispatch is intercepted in turn by the host, which runs the V-guest directly. Consequently, even though there can be guests of guests to almost any depth, from the host's perspective there are only R-guests and V-guests. This bears directly on how VM enables guest exploitation of FP extensions [21].

Real guest support

VM's real guest support for FP extensions has two principal components:

1. *Detection and enablement* detects a guest's attempts to use IEEE FP facilities and establishes the environment to allow that use.
2. *Console functions* support manual operations that affect aspects of a virtual machine's FP extensions.

Detection and enablement

When a real guest is first created and IPLed, it is not enabled to use the FP extensions. That is, the host's AFP-register-control bit (CR0.13) is not turned on. Even though the guest may enable the FP extensions for use by turning on its CR0.13, it is not enabled from VM's perspective until the R-guest issues a BFP instruction or references an AFPR. At that point, VM receives either a BFP-instruction or an AFP-register data exception.

If the guest's CR0.13 is on and FP extensions are installed, VM initializes the additional data structures required to hold the AFPRs and FPC register. It then resumes execution of the guest at the instruction causing the interception, this time with the host's CR0.13 turned on to allow the instruction to be executed by the hardware. Thereafter, whenever the guest stops running (e.g., at the end of its dispatch slice), its AFPRs and FPC register are saved. The next time the guest is dispatched, those registers are restored and the host's CR0.13 is turned on.

The benefit of this implementation is that R-guests that do not exploit the extensions do not require the additional data structures and do not incur the performance cost associated with saving and restoring the AFPRs and FPC register across dispatches.

In the event that a machine check is reflected to a real guest, VM stores guest FPR and FPC register contents in the guest's extended save area if it is defined and enabled.

Console functions

VM Control Program commands (*console functions*) are extended. In particular, the DISPLAY, STORE, and TRACE console functions accommodate the new facilities.

DISPLAY register The DISPLAY command allows the contents of the AFPRs and FPC register to be displayed. Traditionally, displaying an FPR produces the value in hexadecimal and in decimal using scientific notation, interpreting it as an HFP number. DISPLAY works this same way. Since the interpretation of a register's content is program-dependent, the decimal value may not be meaningful in all cases.

The AFPRs and FPC register are not accessible via DISPLAY until the virtual machine has enabled the use of FP extensions, as described in the previous subsection.

DISPLAY storage The DISPLAY command for storage has an I-format option that causes storage contents to be displayed in instruction format. It is extended to display the new floating-point instruction operation code mnemonics and operands.

STORE The STORE command allows the values of the AFPRs and FPC register to be altered. The new contents of an FPR are specified in hexadecimal and may be up to 16 digits long. The FPC register value is specified in hexadecimal and may be up to eight digits long.

The reserved bits of the FPC register must be zero in the value to be stored; otherwise, an error message is produced and the store operation is not performed.

The AFPRs and FPC register are not accessible via STORE until the virtual machine has enabled the use of FP extensions.

STORE STATUS The STORE STATUS command saves the AFPRs and FPC register in an extended save area if one is defined and its use is enabled via guest CR14.2.

TRACE The TRACE command displays the new floating-point instruction operation code mnemonics, operands, and related effective addresses and condition codes when these instructions are encountered.

Virtual guest support
When VM dispatches a V-guest on behalf of an R-guest, the host's CR0.13 is turned on only if both the R-guest's CR0.13 is on and the host has detected that the R-guest is using FP extensions, as described in the subsection on detection and enablement. This means that when a V-guest first uses an AFPR or a BFP instruction, it is retried several times.

The first time the instruction is executed, neither the host nor the R-guest has CR0.13 turned on. The V-guest turns on its CR0.13 and attempts to use one of the new facilities. The host receives a data exception of the appropriate type, observes that the R-guest's CR0.13 is not turned on, and reflects the data exception to the R-guest. The R-guest turns on its CR0.13, enables the V-guest use of FP extensions, and redispatches the V-guest to execute the failing instruction a second time. The host intercepts the dispatch and runs the V-guest on behalf of the R-guest. Again, the instruction causes a data exception because the host's CR0.13 is not turned on. This time, because the R-guest's CR0.13 is on, the host turns on its own CR0.13, enables the R-guest to use FP extensions, and redispatches the V-guest to execute the failing instruction a third (and finally successful) time.

The foregoing description assumes that the R-guest in question is VM/ESA with FP extensions support or a system that behaves similarly. Each additional level of guest causes another instruction re-execution as the enablement process works its way down through the various guest levels until it eventually reaches the host. If any guest along the way does not support FP extensions, its guests and any at higher levels cannot access them.

CMS guest support
CMS, the Conversational Monitor System, is a general-purpose single-user operating system designed specifically to run in a virtual machine. It is mentioned separately here not because it is a different kind of guest from an architectural viewpoint, but rather because there are some subtle considerations for exploiting FP extensions in the CMS environment.

There is currently no specific CMS support for FP extensions. CMS Application programs can exploit the new hardware facilities as long as the underlying hardware and VM/ESA control program support them. However, CMS does not provide new facilities to assist such exploitation. In particular, interrupt handling and multitasking interfaces neither preserve nor restore the AFPRs. A CMS program wishing to exploit FP extensions is responsible for ensuring that the AFP-register-control bit (CR0.13) is turned on.

Despite these limitations, a typical single-user application program running in the CMS environment can take advantage of the new facilities. In fact, initial testing of the VM support was conducted using programs running under CMS.

- *Application support*

Beyond basic enablement, IEEE Floating-Point required enhancements to a variety of application support facilities. These extensions are described below.

Register conventions
The addition of 12 AFPRs and the FPC register required a new S/390 linkage convention to govern how these registers are handled on calls between programs.

Previous conventions
The original S/390 convention was to treat the four FPRs (0, 2, 4, and 6) as volatile across calls.[6]

There is no FPC register in the HFP architecture. The PSW program mask controls whether interruptions occur for certain HFP and other arithmetic exceptions, and in this sense it is analogous to the FPC register. The program mask can be set by programs that establish their own ESPIE routine or by high-level language programs using a run-time service. The PSW program mask is usually assumed to be preserved across calls between programs. Care must be taken when altering the program mask, as this can change the results of some programs. The same considerations hold for changes to the FPC register.

FPR convention
The S/390 linkage convention adopted for 16 FPRs is that FPRs 0 to 7 are volatile and FPRs 8 to 15 are nonvolatile. That is, if a called routine uses any of FPRs 8 to 15, it must save the caller's FPRs before use and restore them before returning to the caller. The called routine may use any of FPRs 0 to 7 without saving and restoring them. To preserve those registers, the caller must save them before the call and restore them afterward. Because FPRs 0–7 are volatile across calls, conventions may be defined that use these registers to return values.

[6] This was more a *de facto* result of not requiring the FPRs to be saved and restored than an explicity stated convention.

Rationale The standard four FPRs were treated as volatile because it was difficult for compilers to optimize by keeping values in these registers, so there was little value in preserving them across calls. With 16 FPRs, significant optimization can be done by keeping values in the FPRs for long periods.

Making some of the FPRs nonvolatile across calls is expected to improve performance by allowing compilers and assembly language programs to leave values in FPRs across calls without having to save and restore them. For example, a calling program can leave data in FPRs 8 to 15, while the called program uses FPRs 0 to 7. The calling program does not have to save and restore FPRs 8 to 15 and, if the called program is a leaf routine, it does not have to save and restore FPRs 0 to 7. (Most serious technical computing platforms have a split of volatile and nonvolatile FPRs.)

An 8/8 split of volatile and nonvolatile FPRs is the simplest convention to document and understand. For compatibility, the original four FPRs must continue to be treated as volatile across a call. Thus, FPRs 0 to 7 are the volatile set.

FPC register convention

The FPC register contents are described in the subsection on the floating-point control (FPC) register. The settings of the IEEE mask and rounding mode bits in the FPC register can affect the results of a program and must be handled carefully.

The majority of applications run with the IEEE defaults: IEEE exceptions are disabled and round-to-nearest is in effect. Some applications enable IEEE exceptions, but only a sophisticated few set a rounding mode other than round-to-nearest.

The S/390 linkage convention is that the FPC register is nonvolatile across calls, except for two fields, the IEEE exception flags and the DXC, which are volatile. Therefore, if a called routine changes any fields in the FPC register other than the IEEE exception flags and the DXC, it must save the caller's values before making the change and restore them before returning to the caller. The called routine can change the IEEE exception flags and DXC, explicitly or by triggering exception conditions, without saving and restoring the caller's values. If the caller depends on maintaining the pre-call value of the IEEE exception flags or DXC, they must be saved before the call and restored afterward.

As with the PSW program mask, it is the responsibility of the application programmer to manage the FPC register settings within the application.

Rationale The IEEE exception mask is analogous to the PSW program mask. This convention is similar to PSW program mask handling and parallels the way AIX handles C programs.

Making the IEEE exception flags and the DXC volatile across a call allows a called routine to use BFP instructions without the overhead of saving and restoring the FPC register.

Components

The component stack that supports applications on OS/390 is made up of language translators (compilers and interpreters), language run-time libraries, a common run-time library, Language Environment (LE) for OS/390, a linker (the DFSMS binder), and two IBM debugger products, *dbx* and the VisualAge Remote Debugger. With the exception of the DFSMS binder, all components are involved in supporting an application's ability to use S/390 IEEE Floating-Point hardware support.

The following sections describe the IEEE Floating-Point support provided by the component stack as well as by other products.

Language translators

Exploitation of IEEE Floating-Point required support in language translators. The related compiler and assembler support is described below.

C/C++ for OS/390 compiler

The OS/390 C/C++ [22–24] compiler was enhanced to support BFP data types and operations. The three floating-point formats supported by the hardware—short, long, and extended—correspond to the types single, double, and double extended, which are defined in the IEEE 754 standard. These in turn correspond to the C and C++ types `float`, `double`, and `long double`.

Compiler support for FP extensions is enabled by the new FLOAT compiler option. This option determines characteristics of the code generated by the compiler, as follows:

- Which floating-point model to use.
- Whether to use the AFPRs (available for both HFP and BFP).
- Whether to use the MULTIPLY AND ADD instruction (for BFP only).

These choices are indicated by keyword suboptions. For example, `FLOAT(HEX,AFP)` specifies that HFP is used and that the compiler can generate code that exploits the AFPRs. This is the recommended setting for "legacy" numeric applications that are to run on a G5 or later processor.

Other new compiler options affect the behavior of the compiler:

ROUND specifies the rounding mode used for BFP computations done at compile time.
STRICT inhibits certain optimizations that may change the order in which computations are done in compiler-generated code.

Details of these options can be found in [22].

In no case does the compiler require a G5 processor during compilation; the user can create applications that exploit the FP extensions on any system running OS/390 V2R6 or later.

It was decided not to provide support for mixing the two floating-point models in the same compilation. The compiler does not provide a means to specify a variable or constant of "the other" floating-point type or to generate instructions belonging to the other model. Furthermore, in C++, the compiler takes explicit steps to avoid the problems that would probably result if compilations using the two models were accidentally mixed: different internal ("mangled") names are used for functions taking floating-point arguments and having the same declared name, depending on the floating-point model used at compile time. This helps prevent the accidental invocation of a function that expects HFP by another, in a different compilation, that passes BFP.

This capability was useful in packaging the C++ Class Library DLLs. Both HFP and BFP versions of many functions were packaged in the same I/O streams DLL. The C++ statement `cout≪1.0;` results in a call to member function `ostream& operator≪(float)` of the `ostream` class. When this statement is compiled with FLOAT(IEEE), the actual ("mangled") name of the function is `_1s_7ostreamFBd`; when compiled with FLOAT(HEX), it becomes `_1s_7ostreamFd`. Having different names for these two implementations of the ≪ operator allows them to be packaged in the same DLL.

C does not offer the function-overloading facilities of C++; a C function is named according to its declared name only, with no decoration coming from its parameter types. To accommodate this situation, the compiler provides a "Feature Test Macro," `_BFP_`, that is defined when FLOAT(IEEE) is specified. This allows developers of libraries to remap names according to the floating-point model in use and package functions that support the two models in the same library (archive) or DLL. One use of this is described in the subsection on C/C++ run-time library floating-point support.

Another change to parameter passing makes it dangerous to try to mix floating-point models in an application, even in C-only code: The "widening" rule for floating-point parameters has changed. In the HFP model, double-precision floating-point numbers can be treated as single-precision simply by using their first four bytes. In both C and C++, HFP single-precision floating-point parameters (`float`) are "widened" to double precision (`double`) when they are passed as parameters. This is not done for BFP parameters, where changing the precision of a floating-point number has greater run-time cost. Thus, a function compiled to use BFP passing two `float`s (single precision) to another function would use eight bytes on the stack, whereas a receiving function compiled to use HFP would expect 16 bytes (two `double`s).

Having the AFPRs resulted in another change in the function-calling mechanism. As discussed in the subsection on register conventions, functions are now expected to save the nonvolatile FPRs if they change their values. The save area provided to a function has not been enlarged to accommodate this; functions that use these FPRs must save them in their own automatic storage in an area described to the run-time by static control blocks compiled into the code. This is described in [25].

High-Level Assembler

The High-Level Assembler (HLASM) [26–29] and its associated Toolkit Feature were updated to support the FP extensions.

HLASM Two major types of enhancement were required in the assembler: new instructions and new data types.

New instructions Many of the FP extensions instructions introduce new instruction formats; these required modifications to the instruction tables and subsequent assembly of symbolic instructions into machine language.

The `ACONTROL` statement provides localized source-program control over HLASM's recognition of the AFPRs:

```
ACONTROL AFPR    Recognize AFPR
ACONTROL NOAFPR  Disallow AFPR
```

`AFPR` means that all 16 floating-point registers are allowed in subsequent statements, while `NOAFPR` means that only FPRs 0, 2, 4, and 6 are allowed.

New floating-point data types Compatibility with floating-point data in existing programs required that they continue to generate the same binary constants. Because there are a limited number of unassigned constant-type letters, we chose a subtype (B, H) *extension* for floating-point constants (types E, D, L), rather than introducing new constant types. For example,

```
DC DB'0.1' long BFP    0.1
DC DH'0.1' long HFP    0.1
```

Using a subtype extension simplifies future language enhancements.

As discussed in the subsection on conversion between decimal and binary floating-point, for typical values the

IEEE standard expects precisely rounded conversion from decimal to binary. Since this required new conversion routines, we used the opportunity to provide support for the four IEEE-defined rounding modes for *both* HFP and BFP. The rounding mode is specified by a new element of the nominal value of a constant:

```
DC DB'0.1 R7' round toward −infinity
DC DH'0.1 R5' round toward 0
```

The new conversion routines also support signed zeros, so that `DC EB' −0'` generates `X'80000000'`.

Consistency is ensured between assembly and run times because the same perfectly rounding algorithms are used in the LE run-time library.

For compatibility, the existing HFP conversions are unchanged. However, conversions with the H subtype provide both rounding controls and slightly greater precision (see examples of "difficult" HFP numbers in the subsection on conversion between decimal and binary floating-point). The conversion routine also supports "traditional" biased round-to-nearest, by adding a correctly signed one-bit to the intermediate result at the position of the most significant discarded bit:

```
DC DH'0.1 R1' traditional rounding
```

The IEEE standard specifies two types of "special value": infinities and NaNs. HLASM supports special values in symbolic format:

- Computational values (INF, MAX, and MIN), as in

  ```
  DC DB'(INF)' long BFP infinity
  ```

  MAX is the largest representable value; MIN is the smallest normalized value of the specified format. (Minimum denormalized values are easily expressible as `XL(n)'1'`, where `n` = 4, 8, or 16.)
- Not-a-Number (SNaN, QNaN, NaN), as in

  ```
  DC DB'(NaN)' long BFP NaN
  ```

  The (`NaN`) format assembles as the default quiet NaN generated by the hardware for invalid-operation exceptions; its two high-order fraction bits are set to `B'10'`. HLASM also supports two software NaNs: a quiet NaN (`QNaN`), with high-order fraction bits set to `B'11'`, and a signaling NaN (`SNaN`), with high-order fraction bits set to `B'01'`.

Some related changes or restrictions are as follows:

- To ensure that no significant bits are lost, restrictions are placed on bit-length modifiers: For computational values in the short, long, and extended formats, the data length must be at least 9, 12, or 16 bits, respectively; the minimum bit lengths for NaNs must be 11, 14, and 18 bits, respectively. (See the discussion of HLAs in the subsection on conversion between decimal and binary floating-point.)
- No scale modifier is allowed for BFP constants, because unnormalized values are not supported.
- No exponent modifier is allowed for special values.
- The allowed range of exponent modifiers and nominal-value exponents was greatly expanded.
- New diagnostics indicate exceeded ranges, invalid modifiers, and invalid rounding indicators.

Because there is no assembly-time support for IEEE standard-defined exception traps or flags, certain predefined actions are taken, depending on the specified (or default) rounding mode:

- Excessively large values round to MAX or infinity.
- Excessively small values round to denormals or zero.

These actions are accompanied by warning messages.

HLASM Toolkit Feature Three components of the Toolkit Feature were enhanced: the disassembler (ASMDASM), the Interactive Debug Facility (ASMIDF), and the Program Understanding Tool (ASMPUT). The enhancements are described in the Toolkit Feature publications.

ASMDASM "Disassembly" is a function common to ASMIDF and ASMDASM. Because disassembling all of the new operations involved extensive modifications, we created a common disassembly module used by both components. This module is also shared by other OS/390 components.

ASMIDF Enhancements to the Debug Facility include the following:

- A new `AFPR` command controls the display of either the traditional four FPRs or all 16 registers plus the FPC register. The contents of displayed registers may be modified by overtyping their contents.
- Storage fields may be displayed in disassembled or "dump" format and may be modified by overtyping.

ASMPUT The Program Understanding Tool was updated to recognize the new operation codes during program analysis.

Assembler summary Many other useful features and enhancements were delivered with HLASM V1R3, and are described in the product documentation [26].

FORTRAN
The primary IBM S/390 FORTRAN Compiler is the VS FORTRAN product. It supports the FORTRAN 77 standard. At the moment, the compiler and its run-time library have not been enhanced to support BFP.

Extensions to the FORTRAN standard have led to the current FORTRAN 90 level of the standard, and a 95 level is anticipated. The FORTRAN 90 standard defines the REAL data type as an IEEE float. To date, the only FORTRAN 90 implementation for OS/390 is provided by a compiler developed by Numeric Algorithms Group, a firm in Oxford, England. At the moment, this compiler emulates IEEE Floating-Point.

Application program run-time libraries
For application programs, run-time support is required to enable effective use of IEEE Floating-Point.

Language Environment
Most of the changes to Language Environment (LE) [25, 30] necessary to support the FP extensions are in the C/C++ language-specific component. However, some changes were required to the base LE component, the Common Execution Library (CEL). These changes can be functionally grouped into two categories: exception handling and saving and restoring nonvolatile registers.

Exception handling In S/390, exceptions are presented to the operating system as program interruptions, which in turn are processed by the operating system and presented to LE via the ESPIE or ESTAE mechanism. LE maps the interrupt code into an "LE condition token," which is the basic construct it uses to represent any exception (software and hardware) to the application (or to a language-specific component, which can map the condition token into a construct appropriate for that language, such as a signal in the C language). For example, a program interrupt code of `X'07'` (data exception) is mapped to the LE condition token CEE3207 (which in the case of C is in turn mapped to a SIGFPE signal).

Interrogating the data exception code and setting the appropriate condition token With the addition of BFP, ten new exception conditions were defined. Since each condition is presented as a data exception program interruption, a new field in the EPIE or SDWA is used to identify it as one of the new BFP exceptions. New LE condition tokens are defined to correspond to the various exceptions.

Default condition handling When a condition is not handled by the application (or the language-specific run-time), CEL performs some default processing, which includes issuing an error message and terminating the application abnormally.

Saving the machine state When LE saves the state of the machine after an exception occurs, it also saves the AFPRs and the FPC register.

CEEDUMP LE's formatted dump facility, CEEDUMP, has been enhanced to display the contents of the 16 FPRs and the FPC register.

Saving/restoring nonvolatile registers Nonvolatile FPRs must be preserved across a call, based on the conventions described in the subsection on register conventions.

Saving nonvolatile floating-point registers Since LE does not know whether any of the nonvolatile FPRs will be modified by the application, its Create Thread routine saves all of them.

Nested enclaves LE creates a "nested enclave" in association with its use of the OS/390 LINK service. When OS/390 starts a program, it sets the FPRs and the FPC register to zero. However, when that program invokes another via LINK, the FPRs and FPC register are not initialized. In order to ensure a consistent programming model for the application, when LE creates a nested enclave, it zeros the FPRs and the FPC register.

Restoring nonvolatile floating-point registers To complete the process of ensuring the integrity of the nonvolatile FPRs, LE's Destroy Execution Environment routine restores the contents of the nonvolatile FPRs.

Out-of-block goto When an out-of-block goto is performed [e.g., when the C `longjmp()` function is invoked, the contents of the nonvolatile FPRs must be restored. Functions such as C's `setjmp()` use a buffer allocated by the application to save the state (e.g., registers, instruction pointer) that `longjmp()` restores. Since this buffer is allocated by the application, it would have to be increased in size to provide room to save the nonvolatile FPRs. Changing the buffer size would introduce an incompatibility, so instead the nonvolatile FPRs are saved on the stack.

As part of the extension to LE to support volatile and nonvolatile FPRs, all LE-compliant applications must ensure that each of their functions and procedures saves the nonvolatile FPRs on the stack (in their stack frame). They must also generate a format 1 Program Prolog Area (PPA1) that identifies which FPRs were saved and where.

This new PPA1 and stack frame format enable LE to restore the nonvolatile FPRs as it traverses from the stack frame of the invoker of the out-of-block goto [e.g.,

```
#include <_Ieee754.h>
#define _FP_MODE_VARIABLE
#include <foo.h>
int main(int argc, char *argv[]) {
   double x, y;
   ...
   __fp_setmode(_FP_HFP_MODE);
   y = foo(x);
   ...
}
```

Figure 9

Hypothetical run-time library function.

the function that issued `longjmp()`] to that of the target [e.g., the function that issued `setjmp()`].

C/C++ run-time library
Beginning with OS/390 V2R6, the C/C++ compiler generates compile units[7] that use either HFP or BFP format to represent floating-point values. This necessitates support for both formats by functions in the C/C++ run-time library [31]. Since functions cannot determine the formats of floating-point parameters and return values by inspection, the compiler and run-time library must provide identification mechanisms for them to use.

Compile-time function mapping The simplest and most efficient identification mechanism is for the run-time library to provide two implementations of each (floating-point-related) service: one to handle HFP-format and the other to handle BFP-format parameters. This is illustrated below for a hypothetical run-time library header, <`foo.h`>, and function, `foo()`, with prototype:

```
double foo(double);
```

Besides `foo()`, the <`foo.h`> header also contains prototypes for

```
double @@FOO@B(double);
/* Version of foo() to handle BFP values */

double @@FOO@H(double);
/* Version of foo() to handle HFP values */
```

[7] This section uses the abstraction of an application composed of a set of compile units produced by different compilers (e.g., assembler compile units are hand-compiled). These compile units may be bound together, by a linkage editor or binder, or not (e.g., as in the case of an application composed of a set of DLLs, each containing one or more bound compile units).

Moreover, <`foo.h`> uses the new compiler macro, `_BFP_`, to select one of these functions during the preprocessing phase of compilation, as follows:

```
#ifdef _BFP_
  #pragma map(foo, "@@FOO@B");
  /* Version of foo() to handle BFP */
#else
  #pragma map(foo, "@@FOO@H");
/* Version of foo() to handle HFP */
#endif
```

The compiler defines `_BFP_` only when the FLOAT(IEEE) option is used. Hence, the conditional logic shown above tells the preprocessor to map `foo` to `@@FOO@B` when the FLOAT(IEEE) option is specified and to `@@FOO@H` otherwise.

Bypassing compile-time function mapping Although compile-time binding of compile units to HFP- or BFP-specific implementations of functions in the run-time library is efficient, the programmer of a compile unit may not always want this to happen. Therefore, besides recognizing `_BFP_`, run-time library headers also recognize a new feature test macro, `_FP_MODE_VARIABLE`, which prevents compile-time mapping of function and constant names in these headers to HFP- or BFP-specific names. For example,

```
#define _FP_MODE_VARIABLE
#include <float.h>
#include <limits.h>
#include <math.h>
  . . .
```

prevents function and constant names in the run-time library <`float.h`>, <`limits.h`>, and <`math.h`> headers from being mapped to HFP- or BFP-specific names during compilation. This allows a compile unit to call run-time library functions to process either HFP or BFP data, independent of its compilation mode. The sample compile-unit source shown in **Figure 9** illustrates how this is done for hypothetical run-time library function `foo()`.

Regardless of the compile option that is used, defining `_FP_MODE_VARIABLE` before <`foo.h`> prevents the function name (`foo`) from being mapped to `@@FOO@H` or `@@FOO@B` during preprocessing. The `_fp_setmode()` function call sets a thread-specific `fp_mode` flag to tell `foo()` whether floating-point parameter `x` is HFP or BFP format. The `fp_mode` flag for a thread remains unchanged until it issues a subsequent `_fp_setmode()` call.

Run-time function mapping Header <`_Ieee754.h`>, which is new for OS/390 V2R6, contains the prototypes for the `_fp_setmode()` function, associated macros, and a related function, `_isBFP()`; see **Figure 10**. Macros in

```
void_fp_setmode(int mode);
_FP_MODE_RESET    /* Determine mode from ieee_option flag in PPA2 */
_FP_HFP_MODE      /* Set fp_mode to hexadecimal floating-point */
_FP_BFP_MODE      /* Set fp_mode to IEEE floating-point */
int___isBFP(void);
```

Figure 10

Function for _fp_setmode ( ).

```
#include <_Ieee754.h>
double foo(double input) {/* Representative of C/C++ run-time *
                           * library floating-point functions. */
   if (__isBFP())
      return @@FOO@B(input);
   else
      return @@FOO@H(input);
}
```

Figure 11

Structure of a floating-point-related run-time library function.

the list shown after the prototype of `_fp_setmode()` are valid values of the *mode* parameter. The comment to the right of each macro in the list gives its meaning as a *mode* value.

Floating-point-related run-time library functions are generally structured as shown in **Figure 11**. Whether the application calls `foo()` directly or calls some other run-time library routine that eventually results in `foo()` being called, the `_isBFP()` function interrogates the `fp_mode` flag value set by the last call to the `_fp_setmode()` function on the same thread. It returns a value indicating whether the mode is HFP or BFP.

Determining compile-time function mapping When the compiler creates a compile-unit object, it includes a format 2 program prolog area (PPA2) to specify attributes of the object. The PPA2 includes a flag indicating whether it is an application or run-time library compile unit. Also, starting with OS/390 V2R6, the PPA2 contains an `ieee_option` flag that the compiler sets to indicate the compilation mode. When `_isBFP()` is invoked, it searches the stack for the current thread until it finds the application compile unit that invoked some run-time library function with a floating-point parameter and returns the value of the `ieee_option` flag in the PPA2 of this application compile unit.

Mixing compile- and run-time function mapping The programmer must be very careful after using `_fp_setmode()` to set a floating-point mode that corresponds to the FLOAT compile option. The `fp_mode` flag set by `_fp_setmode()` is a run-time flag; it affects only run-time library behavior. More specifically, the compiler knows nothing about the run-time `fp_mode` flag. Only the FLOAT option, which is a compile-time flag, affects the floating-point machine instructions the

```
#include <_Ieee754.h>
#define _FP_MODE_VARIABLE
#include <foo.h>
int main(int argc, char *argv[]) {
   double x;
   float  y; /***** Different from original sample source (above) *****/
   ...
   __fp_setmode(_FP_HFP_MODE);
   y = foo(x);
   ...
}
```

Figure 12

Modified version of function shown in Figure 11.

compiler produces. For example, if a compile unit is compiled with the FLOAT(IEEE) option, the compiler generates BFP machine instructions to perform floating-point operations. This is the case even if the compile unit contains a `_fp_setmode(_FP_HFP_MODE)` call.

Suppose the sample program used above is modified slightly, as shown in **Figure 12**. The program will not work. Function `foo(x)` returns an HFP value. The assignment operation fails if the source is compiled with the FLOAT(IEEE) option because the compiler rounds the double value returned by `foo(x)` to float format, destroying the original double HFP result.

<_Ieee754.h> provides a `_fp_cast()` function that can be used to avoid the specific problem this example illustrates. However, it does not provide functions for other operations such as addition, multiplication, or casting to integer formats. In general, the programmer must avoid using operators on floating-point data whose format conflicts with the FLOAT option specified at compile time.

Avoiding function mapping conflicts Perhaps a safer programming practice is to organize applications so that when a compile unit obtains floating-point data that conflicts with the way it was compiled, it immediately calls another compile unit to handle the data. However, it is still the responsibility of the programmer to call a compile unit only if it is compiled to handle the format of floating-point data being passed to it.

Another alternative is to convert floating-point data to the format that matches the FLOAT option of the compile unit(s) that will operate on it. <_Ieee754.h> provides functions to convert between HFP and BFP format. Source and target values may be of float, double, or long double type. When conversion is not exact, a specified rounding mode is used. <_Ieee754.h> contains macros that define the supported data types and rounding modes.

It should be noted that even though conversion between HFP and BFP avoids problems associated with using `_FP_MODE_VARIABLE` and `_fp_setmode()`, the conversion strategy has problems of its own. That is, conversion may be inexact, nonreflexive, or even impossible. For example, the magnitude of (normalized) HFP double values ranges from 10^{-78} to 10^{75}. However, BFP double values range from 10^{-307} to 10^{308}. If a conversion strategy is adopted, return values from conversion functions which indicate that exponent underflow or overflow has occurred must be heeded.

The bottom line is that it is not easy to mix processing of HFP and BFP data in the same application. This is true whether the application consists of a single compile unit or of multiple compile units generated using different options. Because neither hardware nor software can distinguish HFP from BFP values, the onus falls on programmers to exercise care when developing mixed-floating-point-format applications. This is particularly important when adding FLOAT(IEEE) compile units to existing S/390 applications (whose existing compile units and databases are implicitly HFP).

Additional functions Besides the functions already described, <_Ieee754.h> includes

- A set of functions to access the FPC register. These functions can read and change any FPC register field. They may be used, for example, to enable traps or to change rounding mode.
- The `_fp_level()` function, which indicates the level of FP extensions support, either hardware or floating-point simulation, available at run time. Macros in <`_Ieee754.h`> define its return values.
- The `_chkbfp()` function, which indicates whether the FP extensions have been used on a thread and may be used to avoid unnecessarily activating floating-point simulation in an exception handler.

Up to this point, much has been said about functions in <`_Ieee754.h`> because they are unique to OS/390. Functions in other run-time library headers that have floating-point parameters or return floating-point values now support both HFP and BFP formats. Generally, OS/390 C/C++ run-time library functions emulate their AIX counterparts when processing BFP parameters, unless this conflicts with the IEEE 754 standard. However, there are notable exceptions to this principle:

- AIX math library algorithms are not used. Instead, Sun Microsystems Freely Distributed Math Library (fdlibm) was selected as the OS/390 C/C++ run-time library math library, because it is the preferred Java math library. (*Note:* The run-time library enforces round-to-nearest across calls to fdlibm functions. Application rounding mode, if different, is saved during fdlibm execution and restored before return to the application.)
- The OS/390 family of `scanf()` and `printf()` functions use the conversion algorithms (described in the subsection on conversion between decimal and binary floating-point) for BFP values with magnitudes ranging from 10^{-4966} to 10^{4933}. The results of `scanf()` and `printf()` may not be bit-for-bit compatible with AIX because these algorithms round correctly across the entire range, which exceeds the requirements of the IEEE standard.
- All OS/390 C/C++ run-time library formatted input and output functions support slightly different external representations of IEEE Floating-Point QNaN, SNaN, and infinities than does AIX.

Other supported products

Several other products exploit IEEE Floating-Point, as discussed in the following subsections.

Java

The Java language is an object-oriented language that owes a significant part of its syntax to C and C++. Java source code is commonly compiled into a platform-neutral byte code that is executed by a Java virtual machine. At application run-time, the Java virtual machine interprets each of the byte-code operations in sequence.

An alternative to the interpretation done by the Java virtual machine is a just-in-time (JIT) compiler. The JIT improves performance by translating sets of byte codes rather than individual ones, using a variety of optimization techniques to generate efficient machine code, and caching the translated byte code for reuse. The net effect can be to reduce processor use by a factor of two or more.

IEEE Floating-Point and Java The Java Language Specification [8] specifies that single- and double-precision floating-point values and operations conform to the IEEE 754 standard. In some ways the Java definition is tighter:

1. Floating-point operations do not produce exceptions.
2. The Java language always uses round-to-nearest, except when converting floating-point numbers to integers, in which case it uses round-to-zero.
3. Extended formats are not supported.

Prior to the introduction of G5 hardware, all Java floating-point arithmetic on S/390 was emulated. Java Development Kit (JDK**) level 1.1.6 and above, executing on G5 hardware, can take advantage of native BFP support to reduce application execution time.

The Java product contains two separate DLLs (code bases)—one that emulates floating-point arithmetic and one that uses native BFP instructions. The availability of the FP extensions is detected when the Java virtual machine is started and determines which code base is loaded.

JIT and Java virtual machine floating-point use is similar in both DLLs. Emulation is replaced by C function calls in the Java virtual machine and by generation of native code in the JIT. For operations on float or double data, the JIT generates native BFP instructions. For operations on long (64-bit integer) data with float or double data types, the JIT generates native emulation code.

For math functions, the Java virtual machine calls the C run-time library, which uses BFP instructions. The JIT either generates native code or calls the C run-time library.

Performance The performance of Java programs depends on the density of their floating-point operations. For applications with high floating-point content, performance improvement with the FP extensions can be one or two orders of magnitude. The benefits apply to common Java data types and can produce unexpected

improvements. Floating-point data can also be used implicitly in Java routines, such as those that handle dates. In addition, eliminating emulation code reduces program size.

Migration and interaction Applications written entirely in the Java language will not require change; BFP use will be transparent to them, but the interaction of Java programs with existing code that contains calls to floating-point emulation routines will see no performance benefit. For example, extensions can be made to Java applications using the Java native interface (JNI) to invoke code written in other languages. If that non-Java code currently invokes floating-point emulation routines, those calls will continue to be made and application performance will not change.

High-Performance Compiler for Java (HPCJ) The HPCJ is an alternative to interpreting Java byte code using the Java virtual machine. It takes as input Java source code or byte code and then uses the same compiler-based optimization technology as today's C and C++ compilers to generate executables.

The HPCJ is contained in the IBM product VisualAge for Java, Enterprise Edition for OS/390. Exploitation of the FP extensions by HPCJ is available beginning with Release 2 of that product (available from mid-1999) to improve performance for Java applications that use floating-point. This exploitation includes generation of BFP instructions and data in HPCJ object code. Just as for the interpreted Java language, the generation of BFP instructions is transparent to the Java application, which runs unchanged.

Domino 5.0 on S/390

Domino is a client/server application that supports a variety of platforms, including OS/390, OS/400, AIX, Sun Microsystems SunSoft Solaris**, and Windows NT**. In many cases, customers run several Domino servers on different platforms. Domino databases may be copied between servers. Domino also supports replication of databases, which keeps multiple copies of a database on different servers up to date.

Databases may have fields containing IEEE Floating-Point numbers. Domino applications can read such numbers from a database, perform mathematical operations on them, and store the results back in the database. If the database is moved to another platform, some other operations may be performed on those same IEEE Floating-Point numbers. Consequently, it is important that all platforms provide consistent floating-point operations.

LotusScript LotusScript** defines floating-point numbers using the IEEE Floating-Point range. Since the S/390 HFP representation supports a smaller range (for double-precision values), converting IEEE Floating-Point numbers to HFP, performing mathematical operations on them, and converting them back was not a viable solution.

Domino C API Tool Kit The Domino C APIs define all floating-point parameters as IEEE Floating-Point numbers.

Domino floating-point requirements Domino on S/390 had the following requirements:

1. Support math operations on IEEE Floating-Point numbers stored in databases.
2. Support math operations on IEEE Floating-Point numbers passed on C API calls.
3. Support the IEEE Floating-Point value range.
4. Require as few changes as possible to Domino source code.
5. Allow the same Domino executables to run on both new and old S/390 hardware processors.

Solutions to Domino floating-point requirements The FP extensions support added to OS/390 V2R6 exceeded all Domino floating-point requirements. OS/390 V2R6 supports BFP on the G5 processor and on older processors that do not support FP extensions. This was crucial to meeting Domino's requirement of running the same Domino executables on new and older S/390 processors. The new C/C++ compiler FLOAT(IEEE) option made the transition to BFP extremely easy. It was completely transparent to Domino source code running on other platforms. Most significantly, Domino has produced consistent results performing IEEE Floating-Point math operations running on S/390 under OS/390 V2R6 and on other hardware platforms.

IMS DB

IMS* DB does not define floating-point data types and does not have to distinguish HFP and BFP values. Thus, the existing IMS DB product can support transaction programs that use BFP.

When scheduling a new transaction, IMS reuses the task in a message-processing partition. For transactions that use BFP, the FPC register might not be initialized and should be set to the IEEE default value. This is the responsibility of the application or its run-time library. For LE applications, run-time initialization performs this function for each new transaction.

DB2 UDB for OS/390

DB2 Universal Database* for OS/390 (henceforth DB2) must be able to deal with both HFP and BFP at the same time in order to operate on data currently stored in customer tables and allow new applications to use BFP data. Support for BFP must be equivalent to that for HFP to allow customers to implement applications and business processes using BFP. A phased approach is the likely path to achieve this goal.

Toleration In order to take advantage of BFP as soon as possible, DB2 will provide a PTF to allow Assembler and C/C++ programs that use BFP to seamlessly pass that data to DB2.[8] With this PTF, DB2 will convert BFP values to HFP at the application/DB2 interface. DB2 will continue to process and store floating-point data internally as HFP. As data is retrieved from or passed back to an application that uses BFP, DB2 will convert HFP values to BFP. This support is similar to that provided by the SQLJ and JDBC drivers for Java applications that use SQLJ or JDBC to access DB2. Conversion between BFP and other numeric formats such as integer and decimal is also provided.

DRDA applications that use DB2 UDB for OS/390 as a requestor will be able to pass BFP data to the requestor. This data will first be converted to HFP and then passed to the DRDA server.

The LOAD utility will be enhanced to accept BFP-format data as input. The BFP data will be converted to the proper type (Smallint, Integer, Decimal, or HFP Float) as it is loaded into a DB2 table.

In order to take advantage of the performance improvements provided by the new hardware instructions, DB2's existing IEEE Floating-Point/HFP conversion routines will be changed to use these instructions if they are available. DB2 has had its own conversion routines between IEEE Floating-Point and HFP since Version 2 Release 3. If testing determines that the existing software support is not as precise as the new hardware instructions, an option will likely be devised to allow customers to choose which conversion method to use.

Full support Additional support will be needed to achieve parity between BFP and HFP in DB2. A new data type will be required to allow DB2 to store and manipulate BFP data. Changes to statements such as CREATE TABLE, CREATE PROCEDURE, CREATE FUNCTION, and CREATE DISTINCT TYPE will be needed to allow BFP data to be specified. Support at this level includes basic arithmetic operations, as well as built-in function support [e.g., SELECT CHAR(BFP_VALUE) FROM T1], and the ability to specify BFP in a user-defined type (UDT) or pass BFP data to a user-defined function (UDF) or stored procedure (SP). The ability to cast between BFP and other compatible DB2 data types (Smallint, Integer, Decimal, and HFP Float) will also be needed at this level.

Many other issues must also be resolved. Promotion rules, which define how to perform an operation within a group of related data types, will be affected. For example, adding an INTEGER value to a BFP value will result in a BFP value. But what will adding an HFP value and a BFP value produce? The answer is probably BFP, but an option or new syntax may be required to allow the result format to be specified.

Support for BFP in C/C++ applications that use ODBC to access DB2 is another issue that must be addressed.

A change to the DRDA architecture will be needed to allow DB2 to send BFP data to another DRDA client or server. The DRDA architecture as it is now defined determines, at a platform level, the type of floating-point data (HFP or IEEE Floating-Point) that is transmitted by a client or server. DB2 for OS/390 currently sends HFP data, while DB2 for UNIX, NT, OS/2, and AS/400 all send IEEE Floating-Point data. Changes will be needed on all of these platforms to recognize BFP data from DB2 for OS/390.

Conclusions regarding DB2 DB2 will provide toleration support to allow customers to begin using BFP data in their applications. Additional changes will be required in order to provide full support. A new numeric data type will require many changes and much care to ensure the compatibility and integrity of existing customer data and applications while allowing full exploitation of the OS/390 implementation of IEEE Floating-Point.

Debuggers

Two debuggers, Debug Tool and *dbx*, provide support for IEEE Floating-Point.

Debug Tool support for IEEE Floating-Point Debug Tool [32] supports display of expressions involving BFP variables and literals (except expressions that include function calls). It also supports display and modification of BFP variables and registers. The C/C++ compiler options FLOAT(IEEE) and FLOAT(HEX) generate symbolic information about floating-point variables for use by Debug Tool. Debug Tool supports mixing HFP and BFP floating-point in the same program, even if that produces logical inconsistencies. In the example shown in **Figure 13**, external variable `x2` is interpreted as HFP in `m1.c` and as BFP in `p2.c`, depending on the compile option used.

For floating-point expressions, Debug Tool behaves like the C/C++ compiler. All variables and constants

[8] As other languages such as PL/I and COBOL add support for BFP, it is anticipated that DB2 will provide similar support for them.

```
/* m1.c ( compile with FLOAT(HEX) ) */
#include <stdio.h>
extern void setx2(float f);
float x2 = 2.0;   /* x2 is initialized to the HFP version of 2.0 */
int main()
{
  x2 = 2.0;        /* x2 is set to the HFP version of 2.0 */
  setx2(2.0);     /* this call leaves x2 set with the BFP version of 2.0 */
  printf(" x2 = %f\n", x2);  /* prints a BFP version of 2.0 as if it were HFP */
  return 0;
}
/* p2.c  (compile with FLOAT(IEEE) ) */
extern float x2;
void setx2(float f1)
{
   x2 = f1;        /* x2 is set to the HEX version of 2.0 from main() */
   x2 = 2.0;       /* x2 is set to the IEEE version of 2.0 */
}
```

Figure 13

Example program mixing HFP and BFP.

evaluated by the debugger within the scope of a compile unit are interpreted according to the options that are used to compile it. The rounding mode used by Debug Tool to evaluate BFP expressions is always the IEEE default mode, not the one set by the run-time library.

Debug Tool's support of both HFP and BFP can help find bugs related to inconsistent floating-point modes. For example, in `main()`, before the call to `setx2()`, variable `x2` is interpreted as HFP. Displaying `x2` shows the value `2.0000000E+00`. The type of `x2` is reported as a 4-byte float. Within the scope of `m1.c`, all floating-point variables and constants are interpreted as HFP because it was compiled with the FLOAT(HEX) option.

Immediately after entering `setx2()`, variable `x2` is interpreted as BFP. Displaying `x2` shows the value `1.0000000000000000E+001`. Its type is reported as a 4-byte IEEE float, which is different than before. While the value displayed looks like a debugger error, it is a user error because `x2` contains an HFP version of 2.0 that is being interpreted as BFP by the program. Within the scope of `p2.c`, all floating-point variables and constants are interpreted this way because it was compiled with the FLOAT(IEEE) option.

Debug Tool also supports viewing and modification of the AFPRs. For example, evaluating the expression `%FPR3 = 3.14;` sets the value of FPR 3.

UNIX System Services Debugger (dbx): *debugging mixed HFP/BFP programs* The C/C++ compiler and LE allow a load module to contain multiple object files, of which some use HFP and others use BFP. *dbx* supports this kind of mixed-type floating-point data at both the C/C++ source level, in which the data is stored in variables, and the machine level, in which the data is stored in registers. *dbx* performs the appropriate conversions when assignments or references in expressions are made to floating-point values of different types.

On the basis of the type information provided by the compiler, *dbx* knows whether a variable is HFP or BFP. However, this is not true for registers. Therefore, a separate set of symbols has been created to indicate whether the contents of an FPR are to be treated as HFP or BFP. The user decides which type of data is contained in the FPR and uses the appropriate *dbx* symbol to format it as HFP or BFP. The size of the HFP exponent does not change when the format of the floating-point number is 4, 8, or 16 bytes, so representing all floating-point data using an 8-byte register was an acceptable approximation.

However, since the BFP exponent changes size as the format changes, a precision flag allows the user to set the size of HFP or BFP register symbols.

dbx is a C program and allows LE functions to be used to operate on BFP and HFP data. However, this cannot be based on the way in which LE determines whether a compile unit uses HFP or BFP, since one portion of *dbx* code may have to operate on both types of floating-point data. On the basis of the `_BFP_` compiler flag, a global variable in *dbx* is set so that it can determine how it was compiled. Data in an expression is converted to and operated on using the *dbx* compiled floating-point mode. This approach allows most of the expression processing to remain untouched. The only case in which data is converted to the other floating-point type is when the result of an expression is assigned to a variable or a register of that type. Because of the size differences between HFP and BFP exponents, the preferred *dbx* compile mode is BFP.

Users debugging a program with *dbx* can assign either a BFP or an HFP value to a BFP or HFP variable or FPR in any combination. The values of BFP or HFP variables or the FPRs can be displayed at any time, in any combination, and can consist of expressions that have mixed BFP and HFP operands.

Summary and conclusions

The goal of providing IEEE Floating-Point computing on S/390 for use by new workloads has been achieved with the G5 architecture and OS/390 V2R6 and VM/ESA software support. With a G5 architecture that provides most of the support for compliance with the IEEE 754 standard as a basis, operating system infrastructure, language, run-time, other product, and debugging support for IEEE Floating-Point provide a solid yet affordable base for application exploitation. The OS/390 BCP instruction simulator has succeeded in providing this base independently of the G5 hardware and is available to any program. Additionally, the IEEE Floating-Point capability has been added to S/390 in a very transparent way: Applications that use HFP format are unaffected, system performance is identical when the new capabilities are not used, and the ability to service the system is maintained. To enhance the portability of applications between platforms, considerable compatibility with RS/6000 and AIX has been achieved in the architecture and in C/C++ language and run-time external interfaces.

The hardware support in the G5 and the support for C and C++ programs in OS/390 V2R6 enables porting of new floating-point workloads to OS/390 without having to adjust for the historical difference in floating-point formats. Exploitation of the G5 hardware by applications of the Java language for OS/390 allows real use of floating-point in Java programs without the large performance penalty of the previous software simulation. Also, HLASM support allows the creation of highly tuned subroutines using IEEE Floating-Point on S/390.

Given the IEEE Floating-Point capability provided by the G5 processor and OS/390 V2R6, it is expected that numerous applications, and especially emerging workloads, will choose to exploit IEEE Floating-Point when porting to OS/390. While HFP will continue to be fully supported, over time IEEE Floating-Point will become the format of choice for S/390 because of the widespread acceptance of the IEEE 754 standard in the industry.

Acknowledgments

John Franciscovich of IBM VM Development was responsible for turning the concepts behind VM's support of IEEE Floating-Point into reality. Damian Osisek of IBM VM Development provided valuable guidance for the VM implementation. Peter Relson of IBM OS/390 Development designed and implemented the BCP support, including the instruction simulator.

References

1. D. Stevenson, "A Proposed Standard for Binary Floating-Point Arithmetic," *IEEE Computer* **14,** No. 3, 51–62 (March 1981).
2. W. J. Cody, "Analysis of Proposals for the Floating-Point Standard," *IEEE Computer* **14,** No. 3, 63–68 (March 1981).
3. D. Hough, "Applications of the Proposed IEEE 754 Standard for Floating-Point Arithmetic," *IEEE Computer* **14,** No. 3, 70–74 (March 1981).
4. J. T. Coonen, "Underflow and the Denormalized Numbers," *IEEE Computer* **14,** No. 3, 75–87 (March 1981).
5. W. J. Cody et al., "A Proposed Radix- and Word-Length-Independent Standard for Floating-Point Arithmetic," *IEEE Micro* **4,** No. 4, 86–99 (August 1984).
6. "IEEE Standard for Binary Floating-Point Arithmetic," *ANSI/IEEE Standard No. 754-1985*, American National Standards Institute, Washington, DC, 1985.
7. T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley Publishing Co., Inc., Reading, MA, August 1996.
8. J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Addison-Wesley Publishing Co., Inc., Reading, MA, August 1996.
9. IBM Corporation, *Enterprise Systems Architecture/390 Principles of Operation*, Order No. SA22-7201-05 or later; available through IBM branch offices.
10. E. M. Schwarz, R. M. Smith, and C. A. Krygowski, "The S/390 G5 Floating Point Unit Supporting Hex and Binary Architectures," *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, Adelaide, Australia, April 14–16, 1999, pp. 258–265.
11. G. Slishman, "Fast and Perfectly Rounding Decimal/Hexadecimal Conversions," *Research Report RC-15683*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1990.

12. D. Matula, "The Base Conversion Theorem," *Proc. Amer. Math Soc.* **19,** No. 3 (1968).
13. J. T. Coonen, "Contributions to a Proposed Standard for Binary Floating-Point Arithmetic," Ph.D. Thesis, University of California, Berkeley, 1984.
14. W. D. Clinger, "How to Read Floating Point Numbers Accurately," *Proceedings of the ACM SIGPLAN '90 Conference on Programming Design and Implementation*, 1990.
15. G. L. Steele and J. L. White, "How to Print Floating Point Numbers Accurately," *Proceedings of the ACM SIGPLAN '90 Conference on Programming Design and Implementation*, 1990.
16. IBM Corporation, *OS/390 V2R4.0 MVS Planning: Workload Management*, Order No. GC28-1761-03 or later; available through IBM branch offices.
17. IBM Corporation, *OS/390 UNIX System Services User's Guide*, Order No. SC28-1891-05 or later; available through IBM branch offices.
18. IBM Corporation, *OS/390 UNIX System Services Programming Tools*, Order No. SC28-1904-04 or later; available through IBM branch offices.
19. IBM Corporation, *OS/390 UNIX System Services Command Reference*, Order No. SC28-1892-05 or later; available through IBM branch offices.
20. IBM Corporation, *OS/390 UNIX System Services Programming: Callable Services Reference*, Order No. SC28-1899-05 or later; available through IBM branch offices.
21. D. L. Osisek, K. M. Jackson, and P. H. Gum, "ESA/390 Interpretive-Execution Architecture, Foundation for VM/ESA," *IBM Syst. J.* **30,** No. 1, 34–51 (1991).
22. IBM Corporation, *OS/390 C/C++ User's Guide*, Order No. SC09-2361-03 or later; available through IBM branch offices.
23. IBM Corporation, *OS/390 C/C++ Language Reference*, Order No. SC09-2360-03 or later; available through IBM branch offices.
24. IBM Corporation, *OS/390 C/C++ Programming Guide*, Order No. SC09-2362-03 or later; available through IBM branch offices.
25. IBM Corporation, *Language Environment for OS/390 & VM Vendor Interfaces*, Order No. SY28-1152-05 or later; available through IBM branch offices.
26. IBM Corporation, *IBM High Level Assembler for MVS & VM & VSE Language Reference*, Order No. SC26-4940-02 or later; available through IBM branch offices.
27. IBM Corporation, *OS/390 MVS Programming: Assembler Services Guide*, Order No. GC28-1762-05 or later; available through IBM branch offices.
28. IBM Corporation, *OS/390 MVS Programming: Assembler Services Reference*, Order No. GC28-1910-05 or later; available through IBM branch offices.
29. IBM Corporation, *OS/390 MVS Programming: Authorized Assembler Services Guide*, Order No. GC28-1763-05 or later; available through IBM branch offices.
30. IBM Corporation, *Language Environment for OS/390 & VM Programming Guide*, Order No. SC28-1939-06 or later; available through IBM branch offices.
31. IBM Corporation, *OS/390 C/C++ Run-Time Library Reference*, Volumes 1 and 2, Order No. SC28-1663-04 or later; available through IBM branch offices.
32. IBM Corporation, *Debug Tool User's Guide and Reference*, Order No. SC09-2137-03 or later; available through IBM branch offices.

Received November 9, 1998; accepted for publication July 23, 1999

Paul H. Abbott *IBM Centre for Java Technology, IBM Hursley Park, Winchester, SO21 2JN, United Kingdom (pabbott@uk.ibm.com).* Dr. Abbott is a Software Engineer in the Java Technology Centre, working in the Java for OS/390 porting team. He joined IBM in 1997, moving straight into Java porting on the OS/390 platform. He has worked on several areas of the 390 JVM, including the porting and development of Java floating-point on OS/390, AWT support, security, and code conversion. Dr. Abbott received a B.Sc. degree in computer science from the University of Teesside, England, in 1990 and an M.Sc. degree in applied artificial intelligence from Aberdeen University, Scotland, in 1992. In 1996 he was awarded a Ph.D. by Wolverhampton University, England, on the application of artificial intelligence to analytical chemistry. He is an associate member of the British Computer Society.

David G. Brush *IBM System/390 Division, 522 South Road, Poughkeepsie, New York 12601 (k62cdgbc@us.ibm.com).* Mr. Brush is an Advisory Software Engineer working in OS/390 Language Environment Design and Development. He designed IEEE support for the C Run-time Library and Common Execution Library components of LE and was the team leader for implementation of this support. Mr. Brush joined IBM in Kingston, New York, in 1968. He received an M.S. degree in mathematics from Trinity College.

Clarence W. Clark III *IBM System/390 Division, 522 South Road, Poughkeepsie, New York 12601 (clarence@us.ibm.com).* Mr. Clark currently works in the S/390 e-Business area, focusing on Java. Prior to his current assignment, he spent many years working in a variety of technical and managerial positions related to systems and database performance, analysis, and design. For the past several years his work has focused on systems implications related to object technology and associated application development issues.

Chris J. Crone *IBM Software Solutions Division, Santa Teresa Laboratory, San Jose, California 95141 (cjc@us.ibm.com).* Mr. Crone is an Advisory Software Engineer and team leader for the run-time component of DB2 UDB for OS/390. In 1989 he joined IBM in the National Service Division, where he worked on the L3 change team for DB2. In 1990 he moved to the Software Solutions Division and began working in DB2 development. He has contributed to the Stored Procedures and Outer Join projects in DB2 V4R1, ASCII Server support in DB2 V5R1, and most recently was involved with Object Extensions and User-Defined Functions projects in DB2 V6R1. Mr. Crone received a B.S. degree in computer science from the California Polytechnic State University at San Luis Obispo in 1989. He has received an IBM Invention Achievement Award and several IBM individual and team awards for his work on DB2.

John R. Ehrman *IBM Software Solutions Division, Santa Teresa Laboratory, San Jose, California 95141 (ehrman@us.ibm.com).* Dr. Ehrman received the B.A. degree from Oberlin College and the Ph.D. degree from the University of Illinois, both in physics. After 17 years as a computer mathematician at the Stanford Linear Accelerator Center, he joined IBM in 1983, and has since pursued interests in application development tools, including VS FORTRAN and the High-Level Assembler.

Graham W. Ewart *IBM Software Group, Toronto Laboratory, Toronto, Ontario, Canada (ewart@ca.ibm.com).* Mr. Ewart is a Senior Software Developer in C/C++ Development. He has worked in the computer industry for almost 35 years, 23 at IBM. In 1987 he joined the Application Development Technology Centre to work on design and development of the C Compiler family.

Clark A. Goodrich *IBM System/390 Division, 522 South Road, Poughkeepsie, New York 12601 (cgoodric@us.ibm.com).* Mr. Goodrich is currently a Senior Software Engineer on the Domino for S/390 product. While working for IBM, he has also designed and developed software for the OS/390 UNIX System Services signal processing component, DPPX Application Peer Peer Networking (APPN), and DPPX CPU management. Before joining IBM, Mr. Goodrich worked as a scientific application programmer using APL and FORTRAN. He has also written real-time software for the National Aeronautics and Space Administration. Mr. Goodrich graduated from Buffalo State College in 1976 with a bachelor's degree in electrical engineering.

Michel Hack *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (hack@us.ibm.com).* Dr. Hack studied at the École Nationale Supérieure des Télécommunications in France (Ingenieur Civil, 1969) and at Project MAC at the Massachusetts Institute of Technology (M.S. 1972 and Ph.D. 1976), before joining IBM as a Research Staff Member. He started in the 801 group, and moved on to System Programming and Operating System research, where he participated in the development of an experimental operating system for the S/370. He has tracked the evolving S/370 architecture through a machine-level debugger for operating systems that is independent of the software and could thus be used to debug several very different operating systems at the lowest level. Dr. Hack has worked on the low-level software structure of a large-memory fault-tolerant computer system that supports reliable applications across operating system failures. He is currently tracking the evolution of the S/390 architecture into the next century.

John S. Kapernick *IBM System/390 Division, 522 South Road, Poughkeepsie, New York 12602 (kapernic@us.ibm.com).* Mr. Kapernick is a Senior Technical Staff Member in the S/390 Software Systems Design organization. He joined IBM in 1966 in the Data Processing Division in Atlanta, Georgia, working in a variety of field and staff assignments. Since moving to Poughkeepsie, New York, in 1980, he has worked in a variety of technical and management assignments. His current responsibilities are in the area of applications and application-enabling facilities, including development tools for the OS/390 UNIX environment. Mr. Kapernick received a B.I.E. degree from the Georgia Institute of Technology in 1963 and an M.S. degree in system engineering from Polytechnic University in 1994. He has received a number of IBM formal and informal awards, has one patent, and is a member of ACM and the IEEE Computer Society.

Brian J. Minchau *IBM Software Solutions Division, Toronto Laboratory, Toronto, Ontario, Canada (minchau@ca.ibm.com).* Dr. Minchau is a Software Engineer in the IBM Toronto Laboratory. His present responsibilities include language

support for Java, C++, and C within Debug Tool. Since joining IBM in 1991, he has also worked on the C/C++ compiler and prelinker and has an IBM patent for Dynamic Link Library support on OS/390. He received a B.Sc. degree in mathematics from the University of Alberta in 1980, and a Ph.D. degree in theoretical physics from Brown University in 1986.

William C. Shepard *IBM System/390 Division, 522 South Road, Poughkeepsie, New York 12601 (shep@us.ibm.com).* Mr. Shepard is an Advisory Software Engineer in the S/390 Software Systems Design Department. His current responsibilities focus on CPU-related operating system design. Assignments have included operating system performance and constraints analysis, and I/O-related operating system design. He received a B.S. degree in mathematics and physics from St. Lawrence University in 1975, and an M.S. degree in computer science from Villanova University in 1980.

Ronald M. Smith, Sr. *IBM System/390 Division, 522 South Road, Poughkeepsie, New York 12602 (rmsmith1@us.ibm.com).* Mr. Smith is a Senior Technical Staff Member in the Systems Architecture Department of the Mid-Hudson Valley Development Laboratory in Poughkeepsie, New York. He received his B.E.E. degree in electrical engineering from Ohio State University in 1957 and joined IBM at the Endicott Laboratory the same year, moving to Poughkeepsie in 1961. He worked on assignments in circuit design, central processor design, and programming before joining the Systems Architecture Department in 1966.

Richard Tallman *IBM System/390 Division, 522 South Road, Poughkeepsie, New York 12601 (rtallman@us.ibm.com).* Mr. Tallman is a Senior Software Engineer in the S/390 Software Systems Design Department, responsible for the overall design of the Language Environment component of OS/390. He joined IBM in 1974 in Kingston, New York, where he worked on the development of several IBM operating systems and on numerous components within them. In 1990 he was one of the initial designers/developers on the team that added UNIX support to the MVS operating system. In 1996 he moved to Poughkeepsie, New York, to work on OS/390. Mr. Tallman received a B.S. degree in electrical engineering and an M.S. degree in computer engineering from Lowell Technological Institute in 1974. He is the author of a number of patents and has received several IBM Invention Achievement Awards.

Steven Walkowiak *IBM System/390 Division, 522 South Road, Poughkeepsie, New York 12601 (swalk@us.ibm.com).* Mr. Walkowiak is an Advisory Software Engineer in the OS/390 Kernel/dbx Development Department. He is responsible for the overall design of the dbx debugger component of OS/390. Mr. Walkowiak joined IBM in 1981 in Kingston, New York, where he worked on the system test and development of other IBM operating systems and on various components within these operating systems. In 1990 he was one of the initial designers/developers on the team that ported the dbx debugger to the MVS operating system. Mr. Walkowiak moved to Poughkeepsie in 1966 to continue work on debugging on MVS.

Akio Watanabe *IBM Systems Sales Division, Yamato 2428502, Japan (watana@jp.ibm.com).* Mr. Watanabe joined IBM in 1986 and is currently a Staff Marketing Systems Engineer. He worked on the Java Just-In-Time (JIT) compiler for S/390 at the Tokyo Research Laboratory.

W. Romney White *IBM System/390 Division, Endicott Programming Laboratory, Endicott, New York 13760 (romney@us.ibm.com).* Mr. White is a Senior Software Engineer in VM Development. He joined IBM in 1997 and is currently involved in advanced VM development in support of new hardware architectures. Mr. White received a B.Math. degree (1970) in computer science and an M.Math. degree (1976) in software from the University of Waterloo, Canada. He has received several IBM formal and informal awards and is a member of ACM.