Decimal floating-point support on the IBM System z10 processor

E. M. Schwarz
J. S. Kapernick
M. F. Cowlishaw

The latest IBM zSeries® processor, the IBM System z10™ processor, provides hardware support for the decimal floating-point (DFP) facility that was introduced on the IBM System z9® processor. The z9® processor implements the facility with a mixture of low-level software and hardware assists. Recently, the IBM POWER6™ processor-based System p™ 570 server introduced a hardware implementation of the DFP facility. The latest zSeries processor includes a decimal floating-point unit based on the POWER6 processor DFP unit that has been enhanced to also support the traditional zSeries decimal fixed-point instruction set. This paper explains the hardware implementation to support both decimal fixed point and DFP and the new software support for the DFP facility, including IBM z/OS®, Java™ JIT, and C/C++ compilers, as well as support in IBM DB2® and middleware.

Introduction

Most processors today support fixed-point integers and binary floating point (BFP) [1] in hardware but do not support decimal arithmetic natively. Decimal arithmetic is very important for banking and commercial financial transactions. Because transistor size is shrinking at an exponential rate, it follows that more user features should be included in hardware rather than in software, but surprisingly, today's computers do not support a native decimal fixed-point or floating-point data type, with the notable exceptions of simple pocket calculators and mainframes.

Outside of the scientific community, most people prefer the decimal number system. Furthermore, commercial financial transactions need to be performed in a decimal number system because BFP has rounding errors that occur at binary radix points, which most likely do not correspond to a decimal radix point [2]. A decimal format was needed to support commercial computing. In the past, proprietary formats have gained little acceptance in the industry. Concurrent with our investigation into determining a universal decimal floating-point (DFP) format with multiple companies, an IEEE committee was investigating enhancements to the BFP standard. We joined the committee and helped to direct the effort to include the definition of DFP formats and operations in the IEEE 754-2008 Standard [3]. The hope is to get all computers to support a common arithmetic system that can benefit all users. When scientists publish results, the numbers they publish are usually decimal numbers. There would be no rounding errors in the display of their results, assuming there is enough precision, if they did all their computations in a decimal number system. Commercial computing requires decimal now, and as performance closes on binary, there is a possibility that the scientific world will switch to decimal as well.[1] The new IBM System z10* processor and the IBM POWER6* processor, part of the IBM System p* 570 server, support DFP natively in hardware, although support in software is widely available for other platforms.

This paper describes the IBM zSeries* hardware decimal floating-point unit (DFU), showing its dataflow, common instruction execution, and fixed-point operations. Following this is a description of the software that supports the new hardware, including operating systems (OSs), middleware, programming languages, and application-hosting environments.

[1]Professor William Kahan of the University of California Berkeley (the primary architect of the IEEE 754-1985 Standard for BFP computation and its radix-independent follow-on, IEEE 854) at an IEEE 754-2008 committee meeting, December 2002 at Santa Clara, California. About 25 people were present, including two of the authors of this paper.

0018-8646/09/$5.00 © 2009 IBM

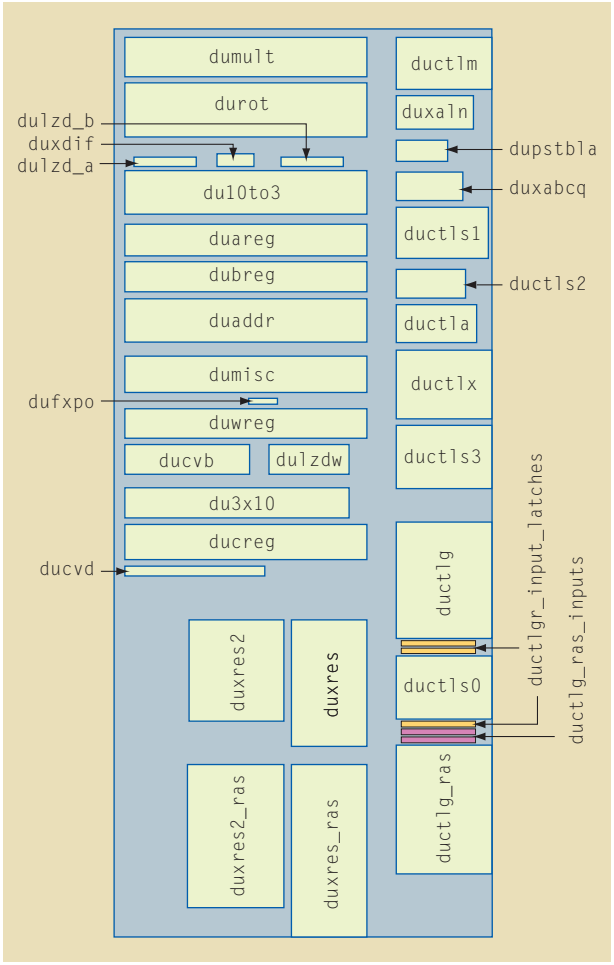

Figure 1

System z10 decimal floating-point unit floorplan.

Decimal floating-point unit hardware

The zSeries DFP facility was introduced in the IBM System z9* platform [4]. It is implemented with a combination of hardware assists and vertical microcode, called *millicode* [5]. The implementation uses the fixed-point decimal hardware to achieve reasonable performance.

The first hardware implementation of the IEEE 754-2008 floating-point standard [3] using a DFU was on the POWER6 processor [6]. The z10* DFU is based on this design, and both were implemented by essentially the same team. The POWER6 processor design was initiated first and was generally available in 2007 on the System p 570 server. At first, there was a common design point for the two processors, but it later diverged to meet the unique requirements of the z10 processor.

The z10 DFU supports DFP operations in a manner similar to the POWER6 processor and also supports the traditional fixed-point decimal operations that have been part of the architecture of IBM mainframes for more than 40 years [7]. The z10 DFU uses the main dataflow of the POWER6 processor with additional interfaces to the fixed-point unit (FXU) and data cache as well as a completely new set of controls to support the additional instructions in the IBM z/Architecture* platform. Both the z10 DFU and the POWER6 processor DFU have 54 DFP instructions, but the z10 DFU has an additional 13 decimal fixed-point instructions and four hardware-assist instructions. DFP operands have three formats: short, long, and extended, which have 7, 16, and 34 digits of significance, respectively, although arithmetic operations are performed only on the long and extended formats. The operands are loaded from memory into the 16×64-bit floating-point register (FPR) files, which are also shared with the binary and hexadecimal floating-point operations. The FPRs are formatted like general memory locations and do not separate the sign, exponent, and significand. The significand is in a compressed format that is expanded into binary coded decimal (BCD) format.

Basic DFU dataflow

The DFU physical dataflow is shown in **Figure 1**. Each box represents the physical dimension of a component, called a *macro*. On the left-hand side of the DFU is the custom dataflow of the significand. Immediately below are several exponent macros, and on the right-hand side are control and exponent macros. The significand custom dataflow is very close to that of the POWER6 processor design, whose functional dataflow is shown with interconnections by Eisen et al. [6].

At the top of the 144-bit-wide dataflow of the significand is the `dumult` macro, otherwise known as the *multiple creator macro*, which creates two times (2×) and five times (5×) the multiplicand. Just below it is the `durot` macro, otherwise known as the *rotator*, which is used for shifting the significand left or right and has a built-in mask used to zero-out digits, depending on the operation. Next, there are a few small macros, including leading-zero detect macros for each operand, `dulzd_a` and `dulzd_b`, and an exponent difference macro, `duxdif`. The `du10to3` macro is used to expand DFP data. The significand is compressed in densely packed decimal (DPD) encoding [8] and is expanded by the `du10to3` macro to BCD encoding. In the top to middle of the stack are the operand A and B registers, `duareg` and `dubreg`. The adder, `duaddr`, is located next to operand registers to reduce wire length in this critical timing path. The adder can be separated into two 18-digit adders or combined into one 36-digit adder that has a latency of two cycles, but a throughput of one add per cycle. Several multiplexers and fixed shifters are contained in the `dumisc` macro. A BCD result register is contained in the

duwreg macro. The BCD data is compressed back to DPD format in the du3x10 macro and placed in the C register, ducreg, to be sent to the FPR. There is also a macro for converting BCD to binary, ducvb, and the reverse, ducvd, and a macro for detecting the number of leading zeros in the result, dulzdw. These macros make up the dataflow of the significand; parity is used to check the interfaces with other units, and residue-3 checking [9] is used to protect the significand dataflow from transient failures.

Below the significand stack is most of the exponent computation logic in the duxres and duxres2 macros. Note that there are latch buffers above and below these macros to stage delayed signals to duplicate copies of the macros (duxres_ras and duxres2_ras) so that checking is not timing critical.

The stack on the right-hand side is mostly for controls and consists primarily of random logic macros. The multiplication and division controls are in ductlm, the addition controls are in ductla, and miscellaneous instruction controls are in ductlx. The ductlg macro is used to perform decodes of the instruction text and also contains global controls. It is duplicated for reliability (ductlg_ras). The ductls1, ductls2, and ductls3 macros perform miscellaneous operations, such as handling special results (e.g., not-a-number) and implementing a common rounding routine used by all arithmetic operations. The ductls0 macro performs much of the RAS (reliability, availability, and serviceability) checking and reporting. Mixed in with the control macros are duxabcq, which holds the input exponents, and duxaln, which creates a shift amount for alignment of the significands. There is a lookup table for division to prescale the operands by an approximation to the reciprocal of the divisor in dupstbla.

DFP addition

Floating-point addition can be split into three cases:

Case 1—Exponents equal.

Case 2—Shift the operand with the bigger exponent only.

Case 3—Shift both operands.

For case 1, the operands are transmitted to the A and B operand registers, duareg and dubreg. In cycle 1, the operands are expanded from DPD format to BCD format by the du10to3 macro and placed back into the A and B operand registers. In cycles 2 and 3, the operands are added in the duaddr macro, and the result is placed in the W register, duwreg. In cycle 4, the BCD result is converted back to DPD format by the du3x10 macro and placed into the C register, ducreg. The result can then be written back to the FPRs.

Cases 2 and 3 use the rotator in durot to align the significands properly. The shift amounts driven by duxaln are determined by subtracting the exponents in duxdif and by determining the number of leading-zero digits.

All three cases might result in rounding. Rounding is accomplished by adding 1 to the significand in the least-significant digit. This can result in a further shift right of the significand. For case 3, subtraction can result in a shift left one digit.

There are many different execution sequences for addition that are dependent on the data. It takes several cycles to inspect the data and determine which sequence to use. Although it is difficult for the DFU to optimally fill the pipeline and signal the instruction dispatch unit (IDU) eight cycles in advance of ending, at least the hardware hides all of these complexities from the user. This shortfall is discussed more in the section "Optimization of floating-point operations."

DFP multiplication

Floating-point multiplication consists of expanding the DPD significand to BCD format, multiplying the multiplicand with all the digits of the multiplier one digit at a time, and then summing the partial products. Since up to twice as many digits are produced in the result as there are in the input operands, rounding may be necessary to fit into the target format.

The second step, creation of digit multiples of the multiplicand, is implemented in the dumult macro. A doubler and quintupler are used to create easy multiples (1×, 2×, 5×, and 10×) of the multiplicand. The doubler and quintupler are very fast because each digit is independent of other digits and there is no carry propagation. All possible multiples of the multiplicand can be formed by a simple addition or subtraction of two of the easy multiples [10].

The 36-digit adder is specially optimized to speed up 16-digit multiplication. The adder can work as a full 36-digit adder or as two independent 18-digit adders. For 16-digit multiplication, 18 digits is the perfect width for an adder.[2] One half of the adder is used to create a new partial product every cycle. New partial products are formed by summing or subtracting two easy multiples of the multiplicand.

The third step, accumulating the partial products, is a little tricky. The adder is a two-cycle adder, but it is pipelined, and a new add can start every cycle. Given that a new partial product is created every cycle, one cannot simply sum the new partial product to the accumulated

[2]This is so because partial products are 17 digits (1 digit × 16 digits results in 17 digits), and when shifted 1 digit from each other, the sum of two consecutive products is 18 digits. Also, 18 digits is perfect summing pairs of partial products because each pair is 18 digits, and when shifted 2 digits from each other, the least-significant 2 digits can be accumulated directly while the upper 18 digits proceed through the adder.

sum because it takes two cycles to accumulate. Instead, pairs of partial products are first summed together and then accumulated in the overall sum, so this half of the adder is split into even and odd cycles, with even cycles used to create sums of paired partial products and odd cycles used to accumulate paired products with the running sum.

For 34-digit multiplication, the adder remains one big 36-digit adder. Partial products are created every other cycle and accumulated every other cycle with the running sum. Therefore, 16-digit multiplication performs a 1×16-digit multiplication (1 digit multiplied by 16 digits) every cycle, resulting in 16 multiplications and a series of additions to accumulate the product; 34-digit multiplication produces a 1×34-digit multiplication every other cycle, resulting in 34 multiplications and a series of additions. Shifting and rounding cycles also result in additional cycles.

DFP division

The floating-point division algorithm is a nonrestoring radix-10 prescaling method [11]. The most significant digits of the divisor are used to index a lookup table of the reciprocal. Two digits of the approximate reciprocal L, called the *prescale amount*, are read from this table and are used to scale the dividend N and divisor D as follows:

$$
\begin{aligned}
&Q = N/D \\
&L \text{ approx } 1/D \\
&N' = N \times L \\
&D' = D \times L \\
&1.0 \le D' < 1.11 \\
&P_i = P_{i-1} \times 10 - q_{i-1} \times D' \\
&q_{i-1} = (P_{i-1})^{\mathrm{tr}},
\end{aligned}
$$

where Q is the quotient, q is the quotient digit, and P_i is the ith iteration partial remainder and tr is used to indicate truncation to one digit. The next partial remainder is dependent on the current partial remainder minus the quotient-digit guess times the scaled divisor. Thanks to prescaling, the quotient-digit guess is simply the most significant digit of the partial remainder. Thus, the lookup table needs to be consulted only at the beginning of the operation, leaving each subsequent iteration much faster.

The multiplication (the quotient-digit guess) $\times$ (the scaled divisor) is accomplished by storing the multiples between 1 and 5. To save area, the multiples between 6 and 9 are determined by precalculating a second partial remainder with the previous quotient-digit guess plus an additional 1 or minus an additional 1. In the subsequent iteration, the plus or minus 1 in the prior iteration has a weight of 10. Two remainders are calculated each iteration. PA_i is calculated as the normal remainder, and PB_i is calculated with an additional plus or minus 1 to the quotient-digit guess:

$$
\begin{aligned}
PA_i &= P_{i-1} \times 10 - q_{i-1} \times D' \\
PB_i &= P_{i-1} \times 10 - (q_{i-1} \pm 1) \times D'.
\end{aligned}
$$

As an example, to form a multiple of 9, the secondary partial remainder of the prior iteration PB is selected and a quotient-digit guess of -1 is chosen, which is equal to $10 - 1 = 9$.

Prescaling takes approximately 19 cycles to normalize the operands, access the lookup table, and multiply the divisor and dividend by a two-digit reciprocal approximation. Each iteration takes four cycles. It takes one cycle to take the most significant digit of the adder output to the multiple selection macro, another cycle to select the multiple, and two cycles in the adder to perform the subtraction of the partial remainder minus the quotient digit times the scaled divisor. Thus, it takes approximately $19 + N \times 4$ cycles latency, where N is the number of digits of quotients needed. It takes a few additional cycles for rounding.

Decimal fixed-point operations

Decimal fixed-point operations have been in the z/Architecture since its beginning in 1964 [7]. Prior generations of zSeries processors execute decimal fixed-point operations in the FXU [12, 13]. The FXU is in the heart of the processor and, on recent machines, contained two 64-bit dataflows. For the z10 processor, the cycle time, at 15 FO4[3], is much faster than prior generations, and the functionality of the FXU had to be reduced. As a result, most multicycle operations were moved out of the FXU, and the dataflow was optimized for binary add, rotation, and bit logical operations. With the growth in the number of decimal instructions as a result of the addition of floating-point instructions, it made sense to create a separate auxiliary unit to handle all decimal operations and to move them out of the critical center of the processor. This had both benefits and disadvantages. Both data types benefited from a dedicated dataflow with all macros close together, but they had to overcome the inherent delay associated with communicating with units closer to the center of the core.

The z10 DFU implements 13 decimal fixed-point instructions [14], including `add` (AP), `subtract` (SP), `compare` (CP), `multiply` (MP), `divide` (DP), `zero and add` (ZAP), `shift and round` (SRP), `convert to binary` (CVB, CVBY, CVBG), and `convert to decimal` (CVD, CVDY, CVDG). `Test decimal` (TP) is executed in the FXU and `edit` (ED) and `edit and mark` (EDMK) are implemented by millicode. Fixed-point operands are variable in length up to 16 bytes, or 31 digits and a sign, as specified by the length field in the instruction text.

[3] *FO4*, or fanout of 4-inverter delay, is the delay of an inverter driving four equivalent loads.

For fixed-point decimal operations, both source operands and the target are in memory. Most processors today do not support memory-to-memory arithmetic operations because of such difficulties as variable-length handling and the fetching of memory operands. However, zSeries processors are optimal for memory accesses because the execution pipeline for one instruction includes both a memory access and an execution stage, whereas RISC computers require multiple instructions to accomplish the same task. Nevertheless, resolving memory interlock dependencies is a concern. Since the operands are in memory, using the result of a prior operation creates an interlock in memory. If the operations are not spaced apart in time, the load/store unit (LSU) or IDU must compare the full addresses to determine the interlock and somehow bypass the operands. The new decimal floating-point architecture makes dependencies easier and faster to handle because the interlocks are simply in registers. It is easier to bypass from 16 specific registers in the FPUs than to bypass within a huge 64-bit address space in the LSU.

Decimal fixed-point operations have an execution latency advantage over the decimal floating-point operations. The main reason they are faster is that they have fewer execution sequences, and thus, it is easier to determine their cycles of latency. This latency is known prior by the IDU and does not have to be signaled eight cycles in advance. However, variable-length multicycle instructions, such as fixed-point decimal multiply and divide, signal the end of execution eight cycles in advance.

The execution of fixed-point decimal operations is the same as the floating-point decimal counterparts, except that no expansion, compression, or rounding cycles are required. For instance, addition involves loading the memory operands into the A and B registers, adding the operands, and then sending out the result. A little more is involved to handle the sign digit and the condition code. The operands in memory are in packed BCD format, which has BCD numeric digits with one digit of sign on the right-hand (or least-significant) end of the operand. The sign digit is aligned one digit to the left of the guard digit in the adder and is forced to zero so it does not affect the operation. Still, the sign digit is important to determine the effective operation: add or subtract. A magnitude result must be produced just as with floating-point sign magnitude notation. This may involve post-complementation if the bigger operand is incorrectly subtracted from the smaller.

For addition and subtraction, the execution latency is seven cycles for operands of 8 bytes or less and nine cycles for operands with greater length. This includes all special cases, including overflow. The compare operation is even faster because no post-processing of a result is needed; it has a latency of five cycles for 8 bytes or less, and a latency of six cycles for longer.

Although decimal fixed-point operations are faster than floating point, it is thought that floating point will yield higher performance in a financial application because of the following advantages:

- Dependencies are at the register level rather than memory level.
- Inherent scale is maintained.
- Inherent rounding can be applied to any of eight selectable rounding modes.
- It offers a greater range of numbers.

Optimization of floating-point operations

Common cases were optimized in the design. In a pipeline, there is an advantage to gaining as much overlap as possible between the current instruction and the following instruction. To gain the optimal overlap, the IDU must be notified eight cycles in advance. This is difficult to do for all operations and all cases of input data, so instead common cases were examined and, specifically, DFP addition case 1 and DFP compare are discussed.

DFP addition case 1

The most common form of addition is case 1 in which the exponents are equal, but even for this case, there could be rounding (if there is a carryout) or underflow handling for a result below the normalized minimum number $N_{\min}$ if the exception is enabled. These subcases are rare. Case 1 would take only four cycles for the most common case. Thus, for case 1, hardware is implemented to detect whether the exponents are equal, that the sum of the most significant digit of each operand is less than 9, and that neither input is subnormal. The only problem is, it takes four cycles to determine this early end detection and to send a signal external to the DFU. This results in the minimum latency of 12 cycles for back-to-back operations because eight cycles are required to start up the next instruction.

DFP compare

Compare does not have a result to round or the possibility of an underflow. Therefore, it can be optimized with a similar means as addition and results in a shorter latency of 11 cycles for all cases.

Execution latencies

The latency of all operations is shown in **Table 1**. There is a variation in the execution latency depending on whether there is underflow, overflow, or rounding. Also, divide and multiply are optimized to process only the number of significant digits needed. Therefore, a smaller number of

Table 1 Execution times of common operations.

Operation	*Type*	*Cycles required for execution*	
		Double-word operands	*Quadword operands*
Add/subtract	Floating point	12–28	16–31
	Fixed point	7	9
Multiplication	Floating point	16–55	17–104
Division	Floating point	16–119	17–193
Compare	Floating point	11	14
	Fixed point	5	6

significant digits will be faster than a rounded full-precision result.

Overview of z10 DFP hardware

The z10 processor offers a high-reliability DFU that implements both the old decimal fixed-point architecture and the new DFP architecture. The instructions are implemented with an FPU that is separate from the binary and hexadecimal floating-point pipelines and also separate from the dual FXU pipelines. The main dataflow consists of a two-cycle 36-digit adder that can be pipelined every cycle. Both DFP and fixed-point decimal use the same hardware-implemented arithmetic algorithms. The floating-point operations require a few more stages to expand, compress, and potentially align operands, round the result, and invoke special results for exception cases. The floating-point operations have the advantage that the operands are passed in registers rather than memory, which results in faster execution of dependencies. Also, floating-point operations automatically take care of scales and implicitly round directly at the correct decimal radix point. The z10 processor offers highly automated and reliable DFP operations.

Software support

Software support for the DFP feature was introduced in the System z9 platform with IBM z/OS* release 6 at the time of the initial availability of the z9* feature [4]. The initial support that was provided was limited to the z/OS control program, IBM z/VM*, and IBM High Level Assembler (HLASM). Work to support the z9 facilities has also begun for Linux** on System z* and the GCC (GNU** compiler collection) tool chain.

z/OS control program support

The z/OS control program support for DFP was built to utilize the infrastructure provided to support IEEE 754-1985 binary floating-point (BFP) standard. This includes detecting the attempted use of hardware DFP (HDFP) facilities or the additional FPRs, activating the hardware features, providing accessible indications of this use, restoring the application state to reexecute the offending instruction, and initiating the saving and restoring of the extended state required by applications that use DFP.

z/VM

z/VM was updated so that guest OSs could make use of the support provided in the System z9 platform at the initial availability of the hardware, although z/VM itself does not make use of DFP.

HLASM support

Support was provided by HLASM for all of the new HDFP instructions and data formats, beginning with the initial availability of the hardware feature in z9 Enterprise Class and Business Class machines. The support was delivered as service updates to the then current releases of HLASM. HLASM allows definition of constant data in all of the DFP data formats, with rounding performed according to the rounding mode requested by the programmer. Programs written to use DFP on z9 processors will run unaltered with improved performance in the z10 processor.

Linux on System z kernel

The Linux on System z kernel takes a different approach to support. An administrative action to update a file, `/proc/cpuinfo`, is done to indicate a DFP-capable central processing unit, and the feature is always turned on.

Conversions

Our motivation for adding IEEE 754-1985 BFP to the System z platform did not include an assumption that users would convert application data from hexadecimal floating point to BFP or the reverse. However, the addition of DFP as a new feature for System z applications assumes that customers want the option to convert applications to DFP from other floating-point systems. We support this by the conversion instruction `perform floating-point operation` (PFPO) [14].

Commercial application software

The objective for DFP is that it must be possible to support the inclusion and use of HDFP in commercial applications. This led us to provide support for DFP data and the hardware DFP instructions in the system components that support the development, deployment, and execution of these applications. Providing appropriate support in the OSs and in assemblers is a

base for this objective, but support in other components is required as well.

Software support in z/OS

Languages and debuggers

The most important programming languages used to write commercial applications that are hosted in z/OS include COBOL, PL/I, C/C++, Java**, and HLASM. Support for DFP data backed by hardware facilities is included in z/OS compilers for all of the languages mentioned above with the exception of COBOL.

Datastores

For the purpose of this discussion, datastores can be divided into those that have schema and those that do not. *Schema* is metadata maintained by the datastore that describes the content of the data. Datastores without schema can be used by applications to store and retrieve arbitrary data without enhancement to the datastore itself, assuming appropriate data definitions are available in the programming languages used to implement the applications. This includes floating-point data, which includes DFP data. File systems and datasets supported by IBM access methods can be readily used to store and retrieve DFP data without further enhancement. The same observation applies to the IBM IMS* (Information Management System) database and the IBM VSAM (Virtual Storage Access Method) database supported by IBM CICS* (Customer Information Control System). Datastores with schema require specific enhancement to the datastore to add support for new data types. This is the case with IBM DB2* for z/OS, a critical component in commercial applications for the System z platform. Such support has been added to DB2 and the SQL (Structured Query Language) used to store and retrieve data.

Application-hosting environments

There are a number of application-hosting environments in the z/OS operating system. Environments provided by IBM can support applications that utilize DFP data and computation. A partial list includes Batch, Time Sharing Option (TSO), UNIX** System Services (USS), IMS, CICS, and IBM WebSphere*. CICS has additional considerations.

Middleware

DFP data is added as a fully supported DB2 data type, DECFLOAT, which is the built-in data type for z/OS version 9 [15]. DFP data types are defined in SQL for DB2. DB2 supports the 8- and 16-byte data formats, but not the 4-byte format. The Query Management Facility products for TSO/CICS and for WebSphere also support the DECFLOAT data type.

Language runtime support

IBM Language Environment* [16] for z/OS supports the C Runtime Library. A set of intrinsic functions that perform computations using the HDFP instructions has been provided, beginning in z/OS release 9. The remainder of these functions will be delivered in the future. They will be used and invoked by code generated by the compilers for all of the programming languages except Java. Language Environment also supports the extended state saved and restored by the OS and, for problem determination purposes, provides it to application-hosting environments or directly to problem determination tools. In addition, Language Environment initializes the contents of the FPC (floating-point control) word at the initialization of application programs.

Programming languages

C/C++—The z/OS C/C++ compilers allow the user to generate C/C++ applications that perform appropriate computations on DFP data using the HDFP instructions included with the z10 processor. The support is delivered with the C/C++ compilers that are part of z/OS release 9 [17, 18]. The compiler DFP option of C and C++ enables the data types `_Decimal32`, `_Decimal64`, and `_Decimal128`. The ARCH [17, 18] directive to the compiler, which specifies the instruction set to be used in the generated program, must be set to 7 or more to enable the use of the DFP option.

Java—Java supports decimal data through the BigDecimal class library. On a z10 processor, the Java Just-in-Time (JIT) compiler associated with the Java 6 JVM** (Java Virtual Machine) generates HDFP instructions to implement this class library for existing and new programs. The performance of the parts of an application using the BigDecimal class library running on a z10 platform will improve over the same application hosted on an earlier System z machine more than the basic difference between the instruction execution speed of the machines as a result of the use of HDFP instructions. The magnitudes of those improvements are unpredictable and related to the intensity of use of BigDecimal data.

PL/1—The Enterprise PL/1 compiler has added support for HDFP in release 3.7, which was first shipped in early October 2007. The PL/1 language has had a DECIMAL FLOAT data type since its creation. A new suboption of the FLOAT option is DFP|NODFP. If it is set to DFP, the compiler treats DECIMAL FLOAT data such that it is handled internally using the HDFP data formats and is processed using the new instructions in the hardware. To create programs that take advantage of the new instructions, data formats, and capabilities, existing programs will have to be recompiled using the new suboption and an ARCH level specification of 7. Some

new built-in functions have been added and others have been updated. An accommodation is made for some library functions for which an HDFP version has not yet been provided [19, 20].

Debuggers

Debugger support for HDFP involves being able to display DFP data stored in main storage or in the FPRs. The FPRs represent a challenge, as they are used to contain floating-point data from all three floating-point systems in the System z platform. The debuggers display the data in the format specified by the user or in hexadecimal digits. The debuggers also allow the user to specify that main storage or FPR content is to be changed. The value it is changed to is floating-point-system dependent. DFP formats have recently been added to both The Open Group dbx debugger and the IBM Debug Tool. This support is quite important because of the internal usage in the hardware of the DPD format to encode DFP data.

Dbx—Dbx is the standard UNIX debugger that is shipped with z/OS. It is used to debug programs written in C/C++ or HLASM that are hosted in the UNIX System Services environment within z/OS. Its HDFP support was provided in z/OS release 9.

Debug Tool—Debug Tool is an optional IBM program product made available as a standalone product or as an option with some z/OS compiler program products. It is the debugger used to debug applications compiled by Enterprise COBOL and Enterprise PL/1. It can be used to debug HLASM and XL C or C++ programs. It debugs programs that are compiled with these languages and hosted in many of the z/OS application-hosting environments. This includes batch, TSO, IMS, CICS, and USS. Debug Tool added support for DFP in version 8 release 1, made generally available in October 2007.

Application-hosting environments

Batch—Batch applications written in HLASM, XL C/C++, Enterprise PL/1, or Java can be written to use DFP data and the HDFP instructions. These programs can use the support provided in DB2 version 9 to store in, retrieve from, and query DFP data in DB2.

USS—USS applications written in HLASM, XL C/C++, Enterprise PL/1, and Java can be written to use DFP data and the HDFP instructions. These programs can use the support provided in DB2 version 9 to store in, retrieve from, and query DFP data in DB2.

TSO—TSO applications written in HLASM, XL C/C++, Enterprise PL/1, and Java can be written to use DFP data and the HDFP instructions. These programs can use the support provided in DB2 version 9 to store in, retrieve from, and query DFP data in DB2.

IMS—IMS applications written in HLASM, XL C/C++, Enterprise PL/1, and Java can be written to use DFP data and the HDFP instructions. These programs can use the support provided in DB2 version 9 to store in, retrieve from, and query DFP data in DB2. They can natively store and retrieve data from IMS databases.

CICS—CICS applications written in HLASM, XL C/C++, Enterprise PL/1, and Java can be written to use DFP data and the HDFP instructions if certain compiler options are used and certain precautions taken. For C, C++, and PL/1 programs, the compiler suboption AFP(VOLATILE) must be used. Programs written in HLASM must save the FPRs used and the FPC, as CICS does not save them across CICS application context switches. These programs can use the support provided in DB2 version 9 to store in, retrieve from, and query DFP data in DB2. They can natively store and retrieve DFP data from CICS VSAM databases.

DB2—In addition to its role as a datastore, DB2 hosts applications known as *stored procedures*. Stored procedures may be written in HLASM, XL C/C++, Enterprise PL/1, and Java to use DFP data. HLASM applications must be Language Environment-enabled to run as stored procedures. Stored procedures hosted in DB2 version 9 can store in, retrieve from, and query DFP data in the relational database. Other datastores without schema can also be used by the stored procedures to store and retrieve DFP data.

WebSphere Application Server—WebSphere applications are Java programs. To the degree that they support decimal data, they do it through the BigDecimal class library. This class library supports decimal data using the HDFP instructions provided in z10 processors when compiled by the Java 6 JIT compiler and the appropriate processor is detected. WebSphere application programs can use the support provided in DB2 version 9 to store in, retrieve from, and query DFP data in DB2 through the use of Sun JDBC** (Java DataBase Connectivity) support.

GCC stack for Linux on System z

IBM implements support in the Linux kernel and the GCC stack but does not distribute Linux for System z. Distribution is handled by external companies, specifically Novell and Red Hat. The support for the GCC stack described below becomes available when the distributions become available.

Binutils—The new instructions in Linux were added to binutils 2.18 for support of the assembler, gas, and objdump. This support is expected to become available in Novell SUSE** Linux Enterprise 11 (SLES 11) and Red Hat** Enterprise Linux 6 (RHEL 6) sometime in 2009.

GCC—GCC added support for the new data types, `_Decimal32`, `_Decimal64`, and `_Decimal128` in a release

that was not externalized. The DFP instructions in the z9 processor and the z10 HDFP instructions are included in the GCC 4.3 compiler, which will be shipped in SLES 11 and RHEL 6, but are available for download [21]. The options required to use the instructions are `-march=z9-ec` and `-mhard-dfp`.

Glibc—Glibc implements and contains the intrinsic library functions using HDFP.

GNU Debugger (GDB)—Support for DFP in C has been provided in GDB release 6.8 [22], made available for download in March 2008 [21]. GDB 6.8 will be included in SLES 11 and RHEL 6 Linux distributions.

Summary

The z10 processor is the first mainframe with hardware support for the DFP format in the IEEE 754-2008 floating-point standard. It joins the IBM POWER6 processor-based System p 570 server as the only hardware support available for this format. Commercial applications require a decimal format and will greatly benefit from this new format. The direct hardware support for DFP arithmetic is also backed by extensive support in software for these data types. OSs, debuggers, programming languages, middleware, and application-hosting environments have been enhanced as required to support this new data type. This provides clients with complete stacks to develop and deploy applications that can take advantage of the DFP facilities introduced with the z9 Enterprise Class and Business Class processors and improved in the z10 processor.

Acknowledgments

We acknowledge the work of the DFU logic team consisting of Adam Collura, Steve Carlough, Wen Li, and Mark Erle. We also acknowledge the software support leaders Peter Relson, Levon Stepanian, Ian McIntosh, Peter Elderon, John Ehrman, Steve Ball, Jim Shearer, Steve Walkowiak, Francisco Anaya, Chris Crone, Eva Hu, Ron Smith, and Michel Hack.

References

1. *IEEE Standard 754-1985*, "IEEE Standard for Binary Floating-Point Arithmetic," ©IEEE; see *http://standards.ieee.org/reading/ieee/interp/754-1985.html*.
2. M. F. Cowlishaw, "Decimal Floating-Point: Algorism for Computers," *Proceedings of the 16th IEEE Symposium on Computer Arithmetic Conference*, Santiago de Compostela, Spain, 2003, pp. 104–111.
3. *IEEE Standard 754-2008*, "IEEE Standard for Floating-Point Arithmetic," IEEE, New York, NY, 2008; ISBN 978-0-7381-5753-5.
4. A. Y. Duale, M. H. Decker, H.-G. Zipperer, M. Aharoni, and T. J. Bohizic, "Decimal Floating-Point in Z9: An Implementation and Testing Perspective," *IBM J. Res. & Dev.* **51**, No. 1/2, 217–227 (2007).
5. L. C. Heller and M. S. Farrell, "Millicode in an IBM zSeries Processor," *IBM J. Res. & Dev.* **48**, No. 3/4, 425–434 (2004).
6. L. Eisen, J. W. Ward III, H.-W. Tast, N. Mäding, J. Leenstra, S. M. Mueller, C. Jacobi, J. Preiss, E. M. Schwarz, and S. R. Carlough, "IBM POWER6 Accelerators: VMX and DFU," *IBM J. Res. & Dev.* **51**, No. 6, 663–683 (2007).
7. G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, Jr., "Architecture of the IBM System/360," *IBM J. Res. & Dev.* **8**, No. 2, 87–101 (1964).
8. M. Cowlishaw, "Densely Packed Decimal Encoding," *IEE Proceedings—Computers and Digital Techniques* **149**, No. 3, 102–104 (2002).
9. F. F. Sellers, Jr., M.-Y. Hsiao, and L. W. Bearnson, *Error Detecting Logic for Digital Computers,* McGraw-Hill, New York, 1968.
10. R. K. Richards, *Arithmetic Operations in Digital Computers*, Van Nostrand Company, Inc., New York, 1955.
11. E. M. Schwarz and S. Carlough, "POWER6 Decimal Divide," *Proceedings of the IEEE 18th International Conference on Application-Specific Systems, Architectures and Processors*, Montréal, Québec, Canada, 2007, pp. 128–133.
12. F. Y. Busaba, C. A. Krygowski, W. H. Li, E. M. Schwarz, and S. R. Carlough, "The IBM z900 Decimal Arithmetic Unit," *Proceedings of the 35th Asilomar Conference on Signals*, Systems and Computers, Pacific Grove, CA, 2001, pp. 1335–1339.
13. F. Busaba, T. J. Slegel, S. R. Carlough, C. A. Krygowski, and J. G Rell, "The Design of the Fixed Point Unit for the z990 Microprocessor," *Proceedings of the 14th ACM Great Lakes Symposium on VLSI*, Boston, MA, 2004, pp. 364–367.
14. IBM Corporation, "z/Architecture Principles of Operation," Document No. SA22-7832-06, February 2008; see *http://publibz.boulder.ibm.com/epubs/pdf/dz9zr006.pdf*.
15. IBM Corporation, *DB2 Version 9.1 for z/OS, What's New*, Document No. GC18-9856-01, 2007; see *http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/topic/com.ibm.db29.doc.wnew/dsnwnk11.pdf?noframes=true*.
16. IBM Corporation, Welcome to Language Environment®; see *http://www-03.ibm.com/servers/eserver/zseries/zos/le/*.
17. IBM Corporation, "z/OS V1R9 XL C/C++ Language Reference," Document No. SC09-4815, 2007; see *http://publib.boulder.ibm.com/infocenter/zos/v1r9/index.jsp?topic=/com.ibm.zos.r9.cbc/cbc.htm*.
18. IBM Corporation, "z/OS V1R9.0 XL C/C++ User's Guide," Document No. SC09-4767-06, September 2007; see *http://publibz.boulder.ibm.com/epubs/pdf/cbcug160.pdf*.
19. IBM Corporation, "Enterprise PL/1 for z/OS V3.7 Language Reference," Document No. SC27-1460-07, 2007; see *http://www-1.ibm.com/support/docview.wss?rs=619&context=SSY2V3&dc=DA400&uid=pub1sc27146007&loc=en_US&cs=UTF-8&lang=en&rss=ct619rational*.
20. IBM Corporation, "Enterprise PL/1 for z/OS V3.7 Programming Guide," Document No. SC27-1457-07, 2007; see *http://publibfp.boulder.ibm.com/epubs/pdf/ibm3pg60.pdf*.
21. Free Software Foundation, Inc., Download GNU; see *http://www.gnu.org/software/software.html*.
22. Free Software Foundation, Inc., GDB: The GNU Project Debugger; see *http://www.gnu.org/software/gdb/*.

Received February 25, 2008; accepted for publication June 4, 2008

Eric M. Schwarz *IBM Systems and Technology Group, 2455 South Road, Poughkeepsie, New York 12601 (eschwarz@us.ibm.com)*. Dr. Schwarz is a Distinguished Engineer in zSeries processor development. He received his B.S. degree in engineering science from Pennsylvania State University, his M.S. degree in electrical engineering from Ohio University, and his Ph.D. degree in electrical engineering from Stanford University. He joined IBM at the Endicott Glendale Laboratories working on follow-ons to the 4381 and 9370 computers. At IBM Poughkeepsie, he worked on the G4, G5, G6, z900, z990, z9 109, and POWER6 computers. He led the floating-point units team for all of these computers and was also chief engineer of the z900. He is currently the core architect on the next zSeries processor.

John S. Kapernick *IBM Systems and Technology Group, 2455 South Road, Poughkeepsie, New York 12601 (pd82jskc@us.ibm.com)*. Mr. Kapernick is a Senior Technical Staff Member in the IBM z/OS System Design organization. His current responsibilities are in the area of applications and application-enabling facilities, including development tools for z/OS and the USS environment. He received his B.I.E. degree from the Georgia Institute of Technology, and his M.S. degree in systems engineering from Polytechnic University. He has received a number of IBM awards, has two patents, and is a member of the Association for Computing Machinery and the IEEE Computer Society.

Mike F. Cowlishaw *IBM Software Group, IBM UK Limited, MP8, P.O. Box 31, Birmingham Road, Warwick CV34 5JL, UK (mfc@uk.ibm.com)*. Mr. Cowlishaw is an IBM Fellow and a Visiting Professor at the Department of Computer Science at the University of Warwick. He created the IBM REXX* programming language, first implemented on the VM/CMS OS in 1979. His current interests are in decimal arithmetic in hardware and software including the IEEE 754 revision, the General Decimal Arithmetic Specification, the decNumber open source and commercial implementation of this in ANSI C, and the enhanced BigDecimal class for Java 5, as described in Java SR-13.