Decimal floating-point in z9: An implementation and testing perspective

A. Y. Duale
M. H. Decker
H.-G. Zipperer
M. Aharoni
T. J. Bohizic

Although decimal arithmetic is widely used in commercial and financial applications, the related computations are handled in software. As a result, applications that use decimal data may experience performance degradations. Use of the newly defined decimal floating-point (DFP) format instead of binary floating-point is expected to significantly improve the performance of such applications. System z9™ is the first IBM machine to support the DFP instructions. We present an overview of this implementation and provide some measurement of the performance gained using hardware assists. Various tools and techniques employed for the DFP verification on unit, element, and system levels are presented in detail. Several groups within IBM collaborated on the verification of the new DFP facility, using a common reference model to predict DFP results.

Introduction

Even though more than half of all commercial data is represented in decimal format, the widely used and hardware-implemented binary floating-point (BFP) unit [1] does not perform decimal calculations with sufficient precision. Applications compensate for this by using software to perform decimal arithmetic, resulting in a heavy performance penalty. The introduction of the decimal floating-point (DFP) [2] facility in hardware is expected to dramatically reduce the runtime of such applications.

The DFP architecture was the first System z* project designed for multiple platforms. More than 50 DFP instructions were added to System z9* while the IEEE Standard P754 [3] was still in the discussion phase. For a timely response, it was decided to implement these instructions mainly in millicode [4] while performing only the most basic tasks in hardware. (*Millicode* is the vertical microcode that executes on the processor.)

This paper provides an overview of the DFP implementation and verification process. We describe the use of System z hardware assists and the millicode implementation of the DFP instruction set. The introduction of the DFP format and instructions required the development of new techniques and tools for all levels of functional verification, from the unit through system level. A major component used by all of these tools is the reference model (RefMod). Various verification teams collaborated on writing the RefMod. This paper discusses the details of new techniques employed by different IBM test-generation tools and describes the successful collaboration among verification and test teams. Experimental results on performance and test coverage of the DFP instructions are given.

The remainder of the paper provides background information on the DFP format, a discussion of its implementation in System z9, and a description of the verification tools employed to verify the correctness of the implementation.

Background

A DFP operand consists of a sign bit, an exponent, and a coefficient. The numerical value of the operand is defined as

$$(-1)^{sign} \times coefficient \times 10^{exponent}.$$

DFP numbers are not normalized. Therefore, multiple representations are possible for a single numerical value. For example, 5×10^2 and 500×10^0 represent the same numerical value. A set of representations of a single numerical value is called a *cohort*. To enhance the portability of DFP applications, IEEE Standard P754 [3] defines a *preferred exponent* that is a function of the input exponents for every arithmetic operation. This exponent uniquely determines the representation selected for the

0018-8646/07/$5.00 © 2007 IBM

Table 1 DFP operand descriptions.

Parameter	*Short*	*Long*	*Extended*
Operand length (bits)	32	64	128
Precision (digits)	7	16	34
Minimum unbiased exponent	−101	−398	−6,176
Exponent bias	101	398	6,176
Maximum unbiased exponent	90	369	6,111

Table 2 Encoding and decoding the DFP CF.

Leftmost digit	*Leftmost two bits of biased exponent*		
	0	*1*	*2*
0	00000	01000	10000
1	00001	01001	10001
2	00010	01010	10010
3	00011	01011	10011
4	00100	01100	10100
5	00101	01111	10101
6	00110	01111	10110
7	00111	01111	10111
8	11000	11010	11100
9	11001	11011	11101

final result. DFP operands are available in three formats: short, long, and extended. The parameters of these formats are summarized in **Table 1**.

The DFP facility shares the floating-point registers (FPRs) with the binary and hexadecimal floating-point facilities. Unlike the binary and hexadecimal floating-point operands [1], the DFP operands are stored in a specially encoded format called Densely Packed Decimal (DPD). This format is specified by the IEEE Standard P754 [3] and was originally proposed in [5]. The format condenses the coefficient as compared with the Binary Coded Decimal (BCD) encoding.

The DPD format comprises four fields [6]:

- S = sign (1 bit): 0 for positive and 1 for negative.
- CF = combination field, which consists of five bits in all formats. For finite numbers, the CF field contains the encoding of the leftmost digit together with the leftmost two bits of the exponent. Special bit patterns of this field, 11110b and 11111b, respectively define infinity and NaN (not a number).
- BXCF = biased exponent continuation field, which consists respectively of six, eight, and 12 bits for short, long, and extended. This field represents the remaining bits of the biased exponent. In the case of a NaN, the leftmost bit is set to 1 for a signaling NaN and to 0 for a quiet NaN.
- CCF = coefficient continuation field, which consists respectively of 20, 50, and 110 bits for short, long, and extended. This field represents the remaining digits of the coefficient, encoded in DPD blocks. Each DPD block, called a *declet*, consists of ten bits representing three decimal digits. Note that a BCD representation of three decimal digits requires 12 bits. Since ten bits encode 1,024 different strings, there are 24 redundant strings. Therefore, a few of these three-digit blocks have redundant encoding.

The DFP operands must be decoded before they are used for execution and encoded before they are delivered as results. More details on the encoding and decoding, and on the encoded representation, can be found in the following subsection.

DFP arithmetic operations are carried out as if they first produced an intermediate result correct to infinite precision and unbounded range. The intermediate result is then rounded according to one of the DFP rounding modes defined by the Floating-Point Control Register (FPCR) to produce a final result in the format of the destination operand. If the final result is exact and has more than one possible representation, the representation with the exponent closest to the preferred exponent is selected. If the final result is inexact, the representation with the smallest exponent is selected to allow for minimum loss of precision bits in the coefficient.

Decoding DFP numbers

The CF field can have two special values: NaN is represented as 11111b, and infinity is represented as 11110b. All remaining values represent the leftmost digit of the coefficient and the two leftmost bits of the exponent, as indicated in **Table 2**.

The CCF is divided into declets (i.e., ten-bit groups). Each declet (pqr stu v wxy) is decoded to create three BCD digits (abcd efgh ijkm). The translation from three four-bit BCD digits to a DPD declet is described in **Table 3**. The reverse translation, from a DPD declet to three four-bit BCD digits, is depicted in **Table 4**.

As an example, let us decode the DFP long number 0xABCDEF0123456789.

- The bit representation of the number is 1 01010 11110011 0111101111 0000000100 1000110100 0101011001 1110001001.
- The sign is 1b and therefore the number is negative.

- The CF is 01010b and, according to Table 2, the leftmost digit is 2 and the leftmost two bits of the exponent are 01b.
- The BXCF is 11110011b (eight bits for the long format). Combining the BXCF and the two leftmost bits of the exponent gives a biased exponent of 0111110011b (i.e., 499). The unbiased exponent becomes 101.
- The first declet (pqr stu v wxy) is 0111101111b. From this, vxwst is 111111b. Therefore, according to Table 4, the three BCD digits are 989.
- The second declet (pqr stu v wxy) is 0000000100b. The value of vxwst is 00100b, and the three BCD numbers are defined as 0pqr, 0stu, and 0wxy, which translate to the three digits of 004.
- The rest of the declets produce 434, 259, and 709.

Therefore, the DFP number with biased exponent is $2{,}989{,}004{,}434{,}259{,}709 \times 10^{499}$. Without the exponent bias, the number is $2{,}989{,}004{,}434{,}259{,}709 \times 10^{101}$, or $2.989004434259709 \times 10^{116}$.

DFP implementation

The z9* is the first System z machine to support DFP. Because the DFP standard [3] was not fully defined when the z9 processor was developed, it was decided to add only basic hardware support to the processor and to implement the DFP instructions in millicode. Millicode is the lowest layer of firmware in the System z and is used, among other purposes, to implement complex instructions where a hardware implementation is not feasible, and to add functions to a product after the hardware is finalized. It is a vertical microcode and is written in a subset of the System z assembly language together with millicode-only instructions known as *milli-ops*. (For a detailed introduction to millicode, see [4].)

Although a millicode implementation does not deliver the performance of a hardware implementation, it was expected that the use of dedicated hardware support would result in a speedup by a factor of 10 compared with a pure software implementation. (A pure hardware implementation should yield another performance improvement of a factor of 10 or more [7].) The millicode can access dedicated hardware through a number of new milli-ops.

Hardware support

The millicode uses a private register set called *Millicode General-purpose Registers* (MGRs). System z processors have performed floating-point functions only by hardware for a long time, and the millicode did not even have easy access to the FPRs on prior processors. To access the FPRs, two new milli-ops were added:

Table 3 BCD-to-DPD conversion.

aei	*pqr*	*stu*	*v*	*wxy*
000	bcd	fgh	0	jkm
001	bcd	fgh	1	00m
010	bcd	jkh	1	01m
011	bcd	10h	1	11m
100	jkd	fgh	1	10m
101	fgd	01h	1	11m
110	jkd	00h	1	11m
111	00d	11h	1	11m

Table 4 DPD-to-BCD conversion.

vxwst	*abcd*	*efgh*	*ijkm*
0----	0pqr	0stu	0wxy
100--	0pqr	0stu	100y
101--	0pqr	100u	0sty
110--	100r	0stu	0pqy
11100	100r	0pqu	100y
11110	0pqr	100u	100y
11111	100r	100u	100y

- Extract FPR indirect (EXFDI): Load an MGR from an FPR.
- Set FPR indirect (SFDI): Load an FPR from an MGR.

The FPR number is determined by one of the four possible register indirect tags [4]. During the millicode entry phase for a non-hardwired System z instruction, the register numbers of the relevant GPRs or FPRs (based on the instruction format) are placed in the register indirect tags. Thus, the millicode can simply refer to the first, second, or third register operand of a System z instruction; it is not required to know the actual register numbers.

Typically, due to the encoding of the DFP numbers (as described in the background section), it is not possible to perform DFP operations without decoding the numbers. It is necessary to extract the exponent and coefficient prior to the operation and to encode the final coefficient and exponent again to form the result. Because both decoding and encoding of the DPD format are frequent tasks, the following milli-ops were added:

- *Extract exponent (EXPDR):* Decode the CF and BXCF fields from the source register and write

the exponent as a binary integer to the destination register.
- *Insert exponent (IXPDR):* Update the CF and BXCF fields of the destination register from an integer in the source register.
- *Extract coefficient (EBCDR):* Decode the BXCF and CCF fields from the source register and write the coefficient as a BCD number to the destination register.
- *Compress coefficient (CBCDR):* Write all fields of the destination register using a BCD number in the source register as a coefficient and assuming an exponent of 0 and a positive sign.

Those milli-ops operate on MGRs as source and destination, and most execute in one cycle. They exist in different forms that allow, together with some simple shifting and bit-masking operations, encoding and decoding of all three proposed DFP formats.

The System z processors have always supported arithmetic on packed fixed-point decimal numbers [6]. However, these instructions operate on operands in storage. To allow decimal calculations in millicode without the need to access storage, four further milli-ops were added that perform BCD arithmetic on the contents of millicode work registers:

- Add decimal register (APRR).
- Subtract decimal register (SPRR).
- Multiply decimal register (MPRR).
- Divide decimal register (DPRR).

APRR and SPRR are single-cycle instructions, whereas MPRR and DPRR require multiple cycles. The four milli-ops use the hardware components designed for the System z decimal instructions. Therefore, the DFP instructions benefit indirectly from the speed provided by the fixed-point decimal hardware.

DFP instruction execution

With the hardware support described in the previous section, the execution of a typical DFP instruction becomes straightforward. A DFP source operand is transferred from an FPR into an MGR and is decoded into exponent, coefficient, and sign. On the basis of these fields, the operand is classified as a finite number, infinity, or NaN. For a dyadic operation (i.e., an instruction with two input operands), the second source operand is decoded and classified in the same manner. A branch table handles the possible combinations of source operands.

If one of the source operands is a special value (infinity or NaN), the result is defined by IEEE Standard P754. Otherwise, the required calculation is performed on exponents, coefficients, and signs. This yields an intermediate result (still expressed as an exponent, coefficient, and sign) with additional digits on the right and a greater exponent range than that defined for the DFP operand type. A sticky bit (i.e., the OR of all of the remaining digits generated during the calculation of the intermediate result) is kept for digits that have been shifted out on the far right.

When the result has multiple representations, the coefficient is shifted until the desired exponent is reached. The intermediate result is then rounded according to the DFP rounding mode. The additional digits and the sticky bit are used to determine whether the result is exact, has to be incremented in the least-significant digit, or must be truncated.

This information is also kept in internal status bits. If the exponent is outside the range for the result type (underflow or overflow), one of the following, depending on the DFP rounding mode and the setting of the IEEE masks in the FPCR [6] results—a subnormal number, zero, infinity, or a wrapped result. (The term *IEEE mask* means *mask bits as defined in IEEE P754*, and similarly for other IEEE terms.)

The rounded result, which now has the required number of digits and the correct exponent range, is encoded into DPD format and written into the destination FPR. Finally, the status bits from the rounding process are used to update the IEEE flags in the FPCR and to raise an IEEE exception if necessary.

Sample calculation

Let us assume that the DFP multiply long instruction

```
MDTR 4, 2, 6
```

is to be executed. This instruction operates on three FPRs. The second and third operands (in FPRs 2 and 6) designate the source. The product is to be placed in the first operand (FPR 4).

Let us assume that FPR 2 contains 0x2DFCC1AEB53B3FBB and FPR 6 contains 0x2238000000000000B. These DPD-encoded operands represent the values $+3{,}141{,}592{,}653{,}589{,}793 \times 10^{-15}$ ($= 3.141592653589793$) and $+81 \times 1$ ($= 81$), respectively.

The millicode first moves the second and third operands from their FPRs into two MGRs using EXFDI instructions. It extracts the exponents and coefficients using appropriate forms of the EXPDR and EBCDR instructions. (For simplicity, let us ignore the sign.) We now have, in four separate MGRs, the following values: For the second operand, an exponent of -15 and a coefficient of 0x3141592653589793; for the third

operand, an exponent of 0 and a coefficient of 0x0000000000000081.

The coefficients are represented as BCD numbers and the exponents as binary integers. (For simplicity, let us ignore the exponent bias.) Both operands are classified as finite numbers.

The millicode now adds the exponents and multiplies the coefficients to form the intermediate result. Addition of the exponents is performed with standard System z arithmetic instructions for binary integers. For the coefficients, however, we use the MPRR and APRR instructions.

The multiplication of two 16-digit coefficients results in a 32-digit intermediate coefficient in an MGR pair. The intermediate result has a coefficient of 0x00000000000000254469004940773233 and an exponent of −15. The preferred exponent for multiplication is defined as the sum of the exponents of the source operands, so our intermediate result already has the required exponent. However, it has 18 significant digits, which is two more than can be accommodated by the long DFP format (16 digits).

This intermediate result is therefore shifted right, incrementing the exponent by one for every digit, until it has no more than 16 significant digits. A guard digit (i.e., the next digit calculated in the intermediate result beyond the precision of the input operand) and a sticky bit are kept on the right for rounding. The coefficient of the intermediate result is now 0x2544690049407732, the exponent is −13, the guard digit is 3, and the sticky bit is 1 (indicating that nonzero bits were shifted out on the right beyond the guard digit).

Further assuming that the DFP rounding mode is "round toward +", the coefficient now has to be incremented in the least-significant digit because the guard digit and sticky bit are not both 0. The final result (now including the sign again) is $+2{,}544{,}690{,}049{,}407{,}733 \times 10^{-13}$ (= 254.4690049407733), and this is encoded as 0x2A06C4C684981FB3 using the CBCDR and IXPDR instructions. This value is written into the first operand location (i.e., FPR 4) by an SFDI instruction.

Finally, the millicode handles the IEEE-inexact condition arising from the fact that the delivered result differs in value from the infinite-precision result. If the IEEE-inexact mask in the FPCR is 0, millicode sets the IEEE-inexact flag in the FPCR. On the other hand, if the mask is 1, the millicode raises a data exception.

Performance

To maximize the performance of the DFP implementation, various algorithms were chosen in accordance with the expected patterns of use. For example, DFP add and subtract instructions, whose source operands have equal exponents, are implemented on a fast path. With the un-normalized number format, we expect this to be a much more frequent case than with BFP numbers (e.g., all currency amounts will have two or three decimal digits to the right of the radix point, and therefore an exponent of −2 or −3). In addition, the performance of the millicode was tuned with the aid of a cycle simulator [8]. This allowed measurements to be made and made it possible to change the code frequently while the hardware was still under development. The cycle simulator also assisted in finding hardware errors.

Table 5 DFP executions in cycles.

Operation	*Software*	*Millicode*
Add/subtract	652 to 1,060	100 to 150
Multiply	4,285	150 to 200
Divide	3,617	350 to 400

The execution time of the instructions is heavily dependent on the data; therefore, only approximate numbers can be given here. We compared the required cycles for the long (16-digit) DFP calculations of multiply, divide, add, and subtract with the cycles for a similar implementation in pure software from [7].

As depicted in **Table 5**, the desired speedup of 10 over a pure software implementation is achieved in most cases.

DFP verification

Several IBM teams collaborated on the verification of the new DFP format in System z. The tools for test generation included floating-point test generator (FPgen) [9] for unit-level verification, architecture verification generator (AVPgen) [10] for core-level (element sim) verification, and systems assurance kernel (SAK) [11, 12] for systems-level verification. Traditionally, each of these verification teams developed and maintained its own randomly generated test cases [13–19] and its result-prediction codes. Because a completely new RefMod was needed for the DFP, the effort was shared among all of these tools.

The remainder of this section highlights the challenges in writing the common RefMod. Detailed descriptions of the various techniques and tools used to test the DFP facility are provided.

Common DFP reference model

Because the DFP architecture was completely new, it was expected to require substantial support from the verification teams. The idea of sharing a RefMod among the various tools came in response to the required effort. The fact that significant commonality was expected among various target platforms also justified the

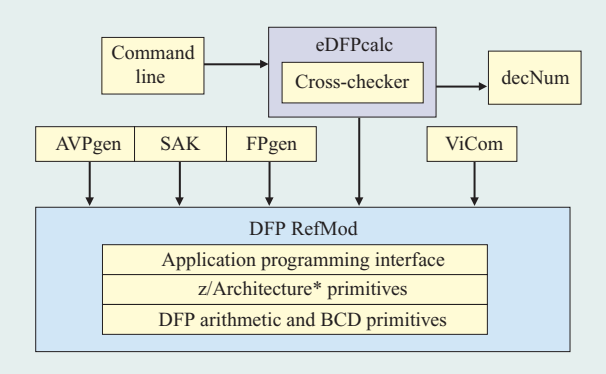

Figure 1

DFP RefMod and the tools connected to it.

development of the RefMod. A new code development methodology was employed, as depicted in **Figure 1**.

The goal was to reduce the System z development and maintenance costs. Future platforms that exploit the DFP architecture were taken into consideration and can easily utilize the RefMod. The challenge was how to develop code at several different sites in parallel while sharing the source code. Decisions were made that the RefMod code should be the following:

- Written in a platform suitable for all of the test tools.
- Modular, easy to maintain, and allowing several parties in different geographical locations to participate in its writing.
- Written in generic code to allow for ongoing updates to specification and possible future development.

The code consists of an internal DFP calculator that performs the actual arithmetic computations for any given decimal format and an interface layer that handles design peculiarities. There were many challenges in developing a common RefMod that could meet the requirements of various platforms and toolsets within a System z platform. Each test tool had a different software library, language (C, C++, PL/X), delivery platform, and operating system (AIX*, Linux**, VM, SAK, and others). Creating a common-source open library was paramount in enabling debug and making quick fixes possible. The methodology was to create a library of embedded source code that utilizes a common language. SAK and ViCom, for example, had to adopt and use C language within their PL/X environments.

This single distributed development team approach creates a potential risk by reducing the independent validations of the architecture to a single point. To mitigate this risk, the work was partitioned into independent subsets with peer reviews, and a unique validation driver was developed to independently cross-check the common verification code. The validation driver (eDFPcalc) uses the decNumber package [20] as an independent means of calculating DFP operations. Additional architectural information was coded on top of this base to provide a cross-checking RefMod (CCheck). The decTest [20] language for describing test cases was extended to handle the architectural components.

The CCheck is capable of generating test cases that can be used as input to the decNumber [20], the RefMod, or both concurrently. By using CCheck, we identified many bugs in the RefMod before the machine was ready to be tested, and thus our confidence in the correctness of the RefMod was increased.

In summary, eDFPcalc provides the following:

- A standalone tool that can perform all DFP instructions. It supports various input data types, such as hex, real, engineering notation, special numbers, and signed or unsigned BCD.
- Control over the rounding mode and the FPCR.
- The capability of running RefMod, decNum, or both.
- Additional command-line options for reporting such entities as errors, error limits, progress, and trace options.
- Display of arithmetic results and architected entities, such as the FPCR.
- Pseudorandom generation of test cases comprising random initial floating-point rounding modes, flags, status, instructions, and input data.
- Result comparison of any combination of the following sources:
  - Results from the decNum package [20] extended with architectural components.
  - Results from the common RefMod.
  - Handwritten expected results.

In addition to directed testing and random testing of the RefMod, regression test suites (consisting of thousands of tests per instruction) were generated to check the proper operation of the RefMod. Each test case in the regression suite is directed to cover some detail of the specification of the instruction or a corner case of the instruction behavior. These cases were defined as constraints on the instruction operands or result, and were run through FPgen (see the FPgen section below) to generate pseudorandom test cases covering these cases. The successful coverage of all cases was measured in terms of coverage metrics (i.e., a fine-granularity partitioning of the test space).

Figure 1 indicates the overall structure of the DFP testing. The RefMod accepts inputs from different

test tools and applications, performs the required DFP operations, and returns the results through the application programming interface (API). The CCheck generates DFP test cases and sends them to the RefMod, decNumber, or both for simulation.

Systems assurance kernel

The systems assurance kernel (SAK), which consists of an OS and a number of system-level test exercisers, has been used to effectively test the architecture compliance of System z machines for more than two decades. Instruction streams consisting of hundreds of machine instructions are created in a pseudorandom manner [11, 12]. Such instruction streams are applied to both the machine under test and SAK result-prediction units. At each interrupt point, the accumulated results of the current test case are compared. An error is reported if any mismatch is detected in the architected resources (e.g., registers, storage, program status word) [11, 12]. SAK is capable of running at the full speed of the machine under test and supports hundreds of copies of different test exercisers running concurrently. As a result, millions of test cases can be generated and verified within a short period of time.

SAK OS and test programs are written in PL/X. To take advantage of the RefMod (which is written in C), SAK was modified so that the SAK PL/X code can interact with the RefMod code. The C executable code contains its own program linkage and stack manipulation. A call stack, which is separate from the original PL/X call stack, was created for C programs. PL/X-to-C call macros and C-to-PL/X call functions provide the linkage between programs that are compiled in two different languages.

The efficiency of pseudorandom test cases depends mainly upon the test-case generation overhead. In DFP testing, selecting meaningful operands may significantly contribute to such overhead. Since the FPRs contain only encoded data [20], substantial time may be spent on generating these operands and encoding them. A means of reducing such overhead is needed to speed up the test process. Enhancing operand interdependency improves the test quality because operand interdependency plays an important role in overall test effectiveness. As mentioned earlier, realizing DFP operand interdependency is complicated because the operands are encoded so that the same operand can be represented in a number of different ways.

SAK employs a method designed to minimize efforts spent on generating meaningful DFP data while enhancing correlations among operands. During the test-case build, the operands can be generated in three ways: randomly, by targeting one of the DFP data classes (with some bias toward the extremes), or from the previously generated operands, as described below.

Once a DFP operand is generated, it is decided at random whether or not to keep it for future use. A new operand, A, can be generated from a previously generated one, B, by modifying a number of bits of B with a simple logical (such as `AND` or `OR`) or shift operation.

The benefit of this is twofold: Because operand A is derived from previously generated operands, operand interdependency, and hence test-case quality, is enhanced. For example, one can easily generate two numbers with the same magnitude but different signs for the same instruction. Second, the effort spent on generating new operands is much less than the effort needed in generating a new operand while enhancing the correlations among the DFP operands. In this case, no decoding or encoding is needed.

When a new operand is generated without going through operand reuse, it is randomly decided whether one of the entries of the lists of previously generated operands can be replaced with this newly generated operand. The fact that the lists are randomly updated enables the method to maintain the dynamic nature of the test-case generation.

Another method used for generating operand data is to form operands from combinations of predefined DPD sets. This step is useful especially when operands of the same instruction are desired to have values that are the same, but they are encoded differently, e.g., in different forms of the redundant DPD.

For experimental purposes, the DFP `ADTR` (i.e., `ADD`) [6] instruction has been run with different operand reuse percentages. For simplicity, when an operand is to be reused, only a single bit of the operand is inverted. Ten copies of a SAK test exerciser were started on logical partition of a z9 machine with five CPUs. Each copy was aimed to run 100 million of the `ADTR` instruction and two load instructions needed for each `ADTR` to load the input operands. Therefore, each copy should run 300 million instructions. The number of times that each of the following cases occurred was monitored:

Case 1: $|\text{Op1}| = |\text{Op2}|$ and $\text{Op1} \neq \text{Op2}$.
Case 2: Op1- and Op2-biased exponents are less than max exponent, and the resultant exponent is greater than max exponent.

The two extremes of the operand reuse show significant improvements in terms of the number of instructions built and executed per minute and the number of times that the rare Case 1 is hit. When 90% of operand reuse is allowed, the test exerciser was generating, executing, and predicting as well as comparing results at a speed of 1,071,428 instructions per minute. On the other hand,

Table 6 Case 1 and Case 2 occurrences and test time (100 million ADTR).

Operand reuse (%)	*Case 1*	*Case 2*	*Duration* (minutes)
0	78	13,680	285.3
10	239	14,368	284.77
20	222	14,860	284.74
30	284	15,421	284.68
40	343	15,372	284.62
50	301	15,394	284.53
60	344	16,050	284.45
70	444	16,789	284.30
80	458	16,954	283.21
90	592	18,140	280.10

the speed was 1,054,370 and 1,051,525 instructions per minute when operand reuse was allowed to be 50% and 0%, respectively. (Even when operand reuse is not permitted, Case 1 can happen, especially when the two operands are +0 and −0.) **Table 6** shows the number of times each Case 1 and Case 2 occurred in different operand reuse percentages. The table also indicates how long it took for each of the test exerciser copies to generate and execute, and predict and compare the results of 300 million instructions (100 million ADTR instructions and their 200 million related load instructions). The slower copies should have taken a little bit longer if the approach of running these copies was slightly modified. Once a copy finishes its runs, the other copies experience fewer CPU contentions and therefore run faster.

AVPgen

AVPgen is a generalized pseudorandom instruction generator used to verify System z architecture [10] on a machine design prior to fabrication [8, 21]. For DFP, AVPgen was used to test the specific milli-ops assists and the machine design in element sim. AVPgen supports an input language for describing the instruction stream and allows various constraints. This language can be generic or specific in detailing instructions, operands, and data values.

AVPgen supports hex and binary floating-point in addition to other System z instructions. The floating-point support includes various heuristics, algorithms, and biasing schemes to generate interesting data values. AVPgen was extended to support DFP and extrapolated Schryer's types [22], Hensel lifting, and Schmookler and Parks algorithms [23] for handling decimal floating-point. Additional extensions for AVPgen for DFP include randomization of possible multiple encodings and preferred encoding (including NaNs) and various cohorts (e.g., forms of zero or numbers with leading or trailing zeros, and infinities). A subset of the biasing scheme used by AVPgen is summarized in **Table 7**.

FPgen

FPgen, a random test generator for floating-point data [24], was initially written for binary floating-point and was extended to support DFP. FPgen supports a powerful input language that allows defining a variety of constraints on the input operands, on the final result, and on the intermediate results. It is possible to define certain relations between input operands and between input and output operands. The input language used by FPgen allows the definition of floating-point scenarios. On the basis of these scenarios, FPgen generates the desired pseudorandom test cases.

A floating-point verification plan is an important complement to the FPgen capabilities as a constraint solver. A generic test plan (GTP) was constructed for floating-point to help engineers with FPU verification. This GTP is based upon experience accumulated during the verification of several processors in IBM along with a deep knowledge and understanding of algorithms and design of the floating-point unit (FPU). The GTP contains many interesting cases to be checked in simulation. It is coverage-oriented, comprising several coverage models, each targeting a specific part of the FPU or a particular feature of floating-point. These models are implemented as input files for FPgen, so that by running FPgen, the user receives pseudorandom test cases that cover the tasks defined in the models.

The input definition language of FPgen allows definition of constraints on DFP numbers as well as on the DPD representation. It is possible to define constraints on the DFP input operand in two ways: on the sign, exponent and coefficient separately, or on the number as a whole.

When constraining each of the fields separately, the sign can be constrained to be plus or minus (or random). The exponent can be constrained to be any single number or any subrange within the legal range. The coefficient can be constrained to be a given mask, in which each digit is constrained to be some subset of the values in $\{0, 1, \cdots, 9\}$. For example, it can be x, indicating that any value is possible; it can be [0–4], meaning the values $\{1, 2, 3, 4\}$; or it can be $\{1, 3, 5, 7, 9\}$, meaning any odd digit.

Constraints on the number as a whole include three possible types: The number can be constrained to have a single value for each of the sign, exponent, and coefficient; it can be constrained to include all numerically equivalent representations of a single decimal value; or it can be a

range between two given values (including all of their equivalent representations). The definition of the constraints applies to the special values infinity and NaN as well, though in a more limited fashion.

The constraint language for the intermediate result contains everything that was specified for the input operands. In addition, the intermediate result has a guard digit and a sticky bit. Constraining the intermediate result enables the definition of many interesting tasks for verification, beyond those defined only through the inputs. For example, it is possible to target specific rounding cases, or many trailing or leading zeros in the output, or cases of overflow.

Special focus was placed on verifying the peculiar attributes of the DFP format compared with the BFP format. For example, to verify the correct implementation of the preferred exponent, we defined a relation constraint between the exponent of the intermediate result and the preferred exponent. With this relation, the user can define constraints such as "The preferred exponent of a divide double precision operation is greater than the exponent of the intermediate result by 30." Only a very rare combination of the inputs will give such a test case.

In addition to defining constraints on the DFP number, constraints on the DPD of the DFP numbers are defined. Since this representation is not very meaningful numerically, we limited our constraints here to include only bit patterns.

The input data generated by DFP constraints is translated to the DPD format for simulation. Because there are several redundant representations in the DPD format, one of the representations is selected at random to provide testing coverage for all representations.

The FPgen GTP was expanded to include a test plan for DFP. We describe a few examples of the various types of models. The main distinction is made between models that are suitable for all instructions and models that are tailored for specific instructions. We give two examples of the former and one example of the latter.

Some of the models aim to cover the entire DFP space. For example, the *basic types model for inputs* contains a list of number types, such as normal number, subnormal number, zero, infinity, and NaN. The model consists of one task for every combination of one input type with another input type.

Other models are directed toward special features of the design. One example is a model whose tasks are all intermediate results in the overflow area. The tasks are denser in the area of overflow that is near the maximal normal value and become sparser as the numbers become larger.

An example of a special model for divide is the model in which the intermediate result is exact and, in addition,

Table 7 AVPgen biasing scheme.

- Full range of exponents biased toward endpoints, numbers biased toward endpoints
  - Selected from minimum exponent to maximum exponent with higher probability of selecting exponents at either end
  - Random selection of fraction digits but biased to endpoints
- Small normalized numbers
- Random leading zeros/max digits, followed by random digit(s) or all zeros/max digit
- Zeros (true, negative, various exponents)
- Small exponent, leading zeros
- One nonzero digit in denorm/subnormal
  - One digit in fraction set to nonzero (1..9)
- Denorm/subnormal
  - Random digits biased toward endpoints
  - Nearby min/max value
- All fraction/coefficient digits either max or nonzero
- N_{min}, N_{max}, D_{min}, D_{max}
  - Exact, random digit changed, nearby
  - Smaller format representation and nearby
- NaNs (random fractions, biased exponent continuation)
- Infinity (random fractions, biased exponent continuation)
- Random normalized
- Schryer types and variants
  - Various coefficient patterns
- DFP specific
  - Random clustering of values that can get nonpreferred encodings
    - Bias toward redundant code points (888 ... 999 tuples)
    - Random forms of infinity/NaNs
    - One digit, max, min, almost max/min
    - Alternate cohort selection
    - Preferred/nonpreferred encoding selection

- Example of specialized biasing per instruction
- Add/sub/compare
  - Exponents in same proximity
  - Equal exponents
  - Equal values
  - Biased random operands (see prior list)
  - One operand zero
  - One digit different
  - Random fraction digit changed, same exponent
  - Result with guard, sticky, and round bits are not zero
  - Toward underflow
  - Toward overflow
  - Leading/trailing/interior zero result
  - Ripple carry/borrow
  - Result CC0/CC1/CC2/CC3 (where CC = condition code)
  - Exact zero difference
  - Varied cohorts
  - Preferred/nonpreferred encodings

we enumerate all possible combinations of leading and trailing zeros in the coefficient of the intermediate result along with all relations between the exponent of the intermediate result and the preferred exponent.

Conclusion

System z9 was the first machine to implement DFP. More than 50 DFP instructions were implemented in millicode with special milli-ops and hardware-assist facilities. Such hardware-assist facilities enabled a speedup factor of 10 over the pure software implementation.

Different verification teams—including SAK, AVPgen, FPgen, and ViCom—collaborated to test the DFP. Each of these teams had to adjust its test strategies to reflect the unique characteristics of the DFP operands. While keeping the verification strategies independent, a common reference model was developed to predict the result of DFP instructions. The use of a common reference brought substantial savings in development and maintenance time. To minimize the risk of using a single reference model, a cross-checker was used for validation. The cross-checker predicts DFP calculations independently on the basis of an external calculator.

Acknowledgments

We thank David Goodman and Ron Mahalik for their contributions to the common reference model. Special thanks go to Mike Cowlishaw for allowing us to reuse the decNumber code, and to Eric M. Schwarz for his support during this project.

*Trademark, service mark, or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

**Trademark, service mark, or registered trademark of Linus Torvalds in the United States, other countries, or both.

References

1. G. Gerwig, H. Wetter, E. M. Schwarz, J. Haess, C. A. Krygowski, B. M. Fleischer, and M. Kroener, "The IBM eServer* z990 Floating-Point Unit," *IBM J. Res. & Dev.* **48**, No. 3/4, 311–322 (2004).
2. M. F. Cowlishaw, "Decimal Floating-Point: Algorism for Computers," *Proceedings of the 16th IEEE Symposium on Computer Arithmetic*, Santiago de Compostela, Spain, June 2003, pp. 104–111; see *http://www2.hursley.ibm.com/decimal/IEEE-cowlishaw-arith16.pdf*.
3. "Draft Standard for Floating-Point Arithmetic," IEEE, Draft 1.2.5, October 4, 2006; see *http://754r.ucbtest.org/drafts/754r.html*.
4. L. C. Heller and M. S. Farrell, "Millicode in an IBM zSeries* Processor," *IBM J. Res. & Dev.* **48**, No. 3/4, 425–434 (2004).
5. M. Cowlishaw, "Densely Packed Decimal Encoding," *IEEE Proc. Computers & Digital Tech.* **149**, No. 3, 102–104 (2002).
6. IBM Corporation, *z/Architecture Principles of Operation* (SA22-7832); see *http://publibz.boulder.ibm.com/epubs/pdf/a2278324.pdf*.
7. M. A. Erle, M. J. Schulte, and J. G. Linebarger, "Potential Speedup with Decimal Floating-Point Hardware," *Proceedings of the 36th Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, CA, November 2002, pp. 1073–1077.
8. B. Wile, M. P. Mullen, C. Hanson, D. G. Bair, K. M. Lasko, P. J. Duffy, E. J. Kaminski, Jr., et al.,"Functional Verification of the CMOS S/390 Parallel Enterprise Server* G4 System," *IBM J. Res. & Dev.* **41**, No. 4/5, 549–566 (1997).
9. Floating-Point Test Generator—FPgen, IBM Corporation; see *http://www.haifa.ibm.com/projects/verification/fpgen*.
10. A. Chandra, V. Iyengar, D. Jameson, R. Jawalekar, I. Nair, B. Rosen, M. Mullen, et al.,"AVPGEN—A Test Generator for Architecture Verification Source," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **3**, No. 2, 188–200 (1995).
11. A. Duale, T. Bohizic, M. Decker, D. Wittig, and G. Darling, "Generation of Pseudo-Random Test Cases," *Proceedings of the 6th World Multiconference on Systemics, Cybernetics and Informatics (SCI)*, Orlando, FL, 2002, pp. 338–341.
12. A. Y. Duale, T. J. Bohizic, and D. W. Wittig, "Pseudo-Random System Testing: Coverage Estimation and Enhancement," *Proceedings of the International Conference on Software Engineering Research and Practice*, Las Vegas, NV, June 2005, pp. 283–289.
13. M. Bailey, T. E. Moyers, and S. Ntafos, "An Application of Random Testing," *Proceedings of the IEEE Military Communications Conference*, November 1995, pp. 1098–1102.
14. B. Beizer, *Software Testing Techniques*, Van Nostrand Reinhold, New York, 1990.
15. T. Y. Chen and Y. T. Yu, "On the Relationship Between Partition Testing and Random Testing," *IEEE Trans. Software Eng.* **20**, No. 12, 977–980 (1994).
16. W. H. Debany, C. R. P. Hatmann, K. G. Mehrotra, and P. K. Varshaney, "Comparison of Random Test Vector Generation Strategies," *Proceedings of the IEEE International Conference on Computer-Aided Design*, Santa Clara, CA, November 1991, pp. 244–247.
17. L. Fournier, Y. Arbetman, and M. Levinger, "Functional Verification Methodology for Microprocessors Using the Genesys Test-Program Generator," *Proceedings of the Conference on Design Automation and Test*, Munich, Germany, 1999, pp. 434–441.
18. M. Karam and G. Saucier, "Functional Versus Random Test Generation for Controllers and Finite State Machines," *Proceedings of Euro ASIC*, Paris, France, 1992, pp. 207–212.
19. S. C. Ntafos, "On Comparisons of Random, Partition, and Proportional Partition Testing," *IEEE Trans. Software Eng.* **27**, No. 10, 949–960 (2001).
20. M. Cowlishaw, "The decNumber C Library," Version 3.37, IBM UK Laboratories, November 22, 2006; see *http://www2.hursley.ibm.com/decimal/decnumber.pdf*.
21. D. F. Ackerman, M. H. Decker, J. J. Gosselin, K. M. Lasko, M. P. Mullen, R. E. Rosa, E. V. Valera, and B. Wile, "Simulation of IBM Enterprise System/9000* Models 820 and 900," *IBM J. Res. & Dev.* **36**, No. 4, 751–764 (1992).
22. N. L. Schryer, "A Test of a Computer's Floating-Point Arithmetic Unit," *Computer Science Technical Report 89*, AT&T Bell Laboratories, 1981.
23. M. Parks, "Number-Theoretic Test Generation for Directed Rounding," *IEEE Trans. Computers* **49**, No. 7, 651–658 (2000).
24. M. Aharoni, S. Asaf, L. Fournier, A. Koifman, and R. Nagel, "FPgen—A Test Generation Framework for Datapath Floating-Point Verification," *Proceedings of the Eighth IEEE International High-Level Design Validation and Test Workshop 2003 (HLDVT03)*, November 2003, pp. 17–22.

Received March 10, 2006; accepted for publication June 28, 2006; Internet publication January 16, 2007

Ali Y. Duale *IBM Systems and Technology Group, 2455 South Road, Poughkeepsie, New York 12601 (duale@ibm.com)*. Dr. Duale received B.E., M.E., and Ph.D. degrees, all in electrical engineering, from the City University of New York. He is currently a Senior Engineer and leads SAK CPU verification. He co-chaired the 18th International Conference on Testing Communicating Systems. He was recognized as a Modern Day Technology Leader at the 2006 Black Engineer Award Conference. Dr. Duale holds a U.S. patent and has published numerous technical papers in the area of testing.

Mark H. Decker *IBM Systems and Technology Group, 2455 South Road, Poughkeepsie, New York 12601 (mdecker@us.ibm.com)*. Mr. Decker received a B.S. degree in electrical engineering from Carnegie Mellon University and an M.S. degree in computer engineering from Syracuse University. He is a Senior Technical Staff Member working in System z architecture verification. He led and contributed to the common DFP reference model development and created the DFP validation driver. Mr. Decker holds several patents and received an IBM Corporate Award for System z architecture verification.

Hans-Georg Zipperer *IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (zipperer@de.ibm.com)*. Mr. Zipperer received a Dipl.-Inform. degree in computer science from the University of Stuttgart, Germany. He initially worked on the System z G3 and G4 CMOS microprocessors, and later worked in processor firmware development for System z, where he was a member of the global millicode team. Mr. Zipperer is currently working on the firmware for the next IBM System z and System p* processors.

Merav Aharoni *IBM Haifa Research Laboratory, Haifa University, Mount Carmel, Haifa 31905, Israel (merav@il.ibm.com)*. Ms. Aharoni received her B.A. and M.A. degrees in computer science from the Israel Institute of Technology (Technion). She has been a Research Staff Member at the IBM Haifa Research Laboratory since 2000. Ms. Aharoni is project leader of FPgen, a deep-knowledge dedicated test generator for floating-point.

Theodore J. Bohizic *IBM Systems and Technology Group, 2455 South Road, Poughkeepsie, New York 12601 (bohizic@us.ibm.com)*. Mr. Bohizic is a Senior Technical Staff Member. He spent more than 25 years developing verification tools for multiple architecture and hardware platforms, concentrating on automatic pseudorandom test generation. He led the IBM SAK effort for many years. Mr. Bohizic's current responsibility includes a technical leadership in the System z emulation project.