INSTRUCTIONS TO ASSIST THE PROCESSING OF A CIPHER MESSAGE

This invention relates to computer system architecture and particularly to the processing of new instructions which augment the IBM z/Architecture and can be emulated by other architectures.

Before our invention IBM has created through the work of many highly talented engineers beginning with machines known as the IBM System 360 in the 1960s to the present, a special architecture which, because of its essential nature to a computing system, became known as "the mainframe" whose principles of operation state the architecture of the machine by describing the instructions which may be executed upon the "mainframe" implementation of the instructions which had been invented by IBM inventors and adopted, because of their significant contribution to improving the state of the computing machine represented by "the mainframe", as significant contributions by inclusion in IBM's Principles of Operation as stated over the years. The First Edition of the *z/Architecture Principles of Operation* which was published December, 2000 has become the standard published reference as SA22-7832-00.

We determined that further new instructions would assist the art and could be included in a z/Architecture machine and also emulated by others in simpler machines, as described herein.

The present invention provides a method as claimed in claim 1.

The features of the preferred embodiments of the invention will be apparent to one skilled in the art from the following detailed description of the invention taken in conjunction with the accompanying drawings in which:

Fig. 1 is the Cipher Message (KM) instruction in the RRE instruction format;

Fig. 2 is the Cipher Message with Chaining (KMC) instruction in the RRE instruction format;

Fig. 3 is a table showing the function codes for the Cipher Message instruction of Fig. 1;

Fig. 4 is a table showing the function codes for the Cipher Message with Chaining instruction of Fig. 2;

Fig. 5 is a representation of the general register assignments for the KM and KMC instructions;

Fig. 6 illustrates the symbol for the Bit-Wise Exclusive Or;

Fig. 7 illustrates the symbols for DEA Encryption and Decryption;

Fig. 8 illustrates the format for the parameter block of the KM-Query;

Fig. 9 illustrates the parameter block of KM-DEA;

Fig. 10 illustrates the KM-DEA Encipher Operation;

Fig. 11 illustrates the KM-DEA Decipher Operation;

Fig. 12 illustrates the format for the parameter block for KM-TDA-128;

Fig. 13 illustrates the KM-TDEA-128 Encipher Operation;

Fig. 14 illustrates the KM-TDEA-128 Decipher Operation;

Fig. 15 illustrates the format for the parameter block for KM-TDEA-192;

Fig. 16 illustrates the KM-TDEA-192 Encipher Operation;

Fig. 17 illustrates the KM-TDEA-192 Decipher Operation;

Fig. 18 illustrates the format for the parameter block for KMC-Query;

Fig. 19 illustrates the format for the parameter block for KMC-DEA;

Fig. 20 illustrates the KMC-DEA Encipher Operation;

Fig. 21 illustrates the KMC-DEA Decipher Operation;

Fig. 22 illustrates the format for the parameter block for KMC-TDEA-128;

Fig. 23 illustrates the KMC-TDEA-128 Encipher Operation;

Fig. 24 illustrates the KMC-TDEA-128 Decipher Operation;

Fig. 25 illustrates the format for the parameter block for KMC-TDEA-128;

Fig. 26 illustrates the KMC-TDEA-192 Encipher Operation;

Fig. 27 illustrates the KMC-TDEA-192 Decipher Operation;

Fig. 28 is a table showing the priority of execution of KM and KMC;

Fig. 29 illustrates our cryptographic coprocessor; and

Fig. 30 shows the generalized preferred embodiment of a computer memory storage containing instructions in accordance with the preferred embodiment and data, as well as the mechanism for fetching, decoding and executing these instructions, either on a computer system employing these architected instructions or as used in emulation of our architected instructions.

The CIPHER MESSAGE (KM) instruction and the CIPHER MESSAGE WITH CHAINING (KMC) instruction will first be discussed, followed by a discussion of the preferred computer system for executing these instructions. In the alternative, a second preferred computer system which emulates another computer system for executing these instructions will be discussed.

CIPHER MESSAGE (KM)

Fig. 1 is the Cipher Message (KM) instruction in the RRE instruction format.

CIPHER MESSAGE WITH CHAINING (KMC)

Fig. 2 is the Cipher Message with Chaining (KMC) instruction in the RRE instruction format.

A function specified by the function code in general register 0 is performed.

Bits 16-23 of the instruction are ignored. Bit positions 57-63 of general register 0 contain the function code. Figs. 3 and 4 show the assigned function codes for CIPHER MESSAGE and CIPHER MESSAGE WITH CHAINING, respectively. All other function codes are unassigned. For cipher functions, bit 56 is the modifier bit which specifies whether an encryption or a decryption operation is to be performed. The modifier bit is ignored for all other functions. All other bits of general register 0 are ignored. General register 1 contains the logical address of the leftmost byte of the parameter block in storage. In the 24-bit addressing mode, the contents of bit positions 40-63 of general register 1 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of general register 1 constitute the address, and the contents of bit positions 0-32 are ignored.

In the 64-bit addressing mode, the contents of bit positions 0-63 of general register 1 constitute the address.

The function codes for CIPHER MESSAGE are shown in Fig. 3.

The function codes for CIPHER MESSAGE WITH CHAINING are shown in Fig. 4.

All other function codes are unassigned. The query function provides the means of indicating the availability of the other functions. The contents of general registers R1, R2, and R1 + 1 are ignored for the query function.

For all other functions, the second operand is ciphered as specified by the function code using a cryptographic key in the parameter block, and

the result is placed in the first-operand location. For CIPHER MESSAGE WITH CHAINING, ciphering also uses an initial chaining value in the parameter block, and the chaining value is updated as part of the operation.

The R1 field designates a general register and must designate an even-numbered register; otherwise, a specification exception is recognized.

The R2 field designates an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost byte of the first and second operands is specified by the contents of the R1 and R2 general registers, respectively. The number of bytes in the second-operand location is specified in general register R2 + 1. The first operand is the same length as the second operand.

As part of the operation, the addresses in general registers R1 and R2 are incremented by the number of bytes processed, and the length in general register R2 + 1 is decremented by the same number. The formation and updating of the addresses and length is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R1 and R2 constitute the addresses of the first and second operands, respectively, and the contents of bit positions 0-39 are ignored; bits 40-63 of the updated addresses replace the corresponding bits in general registers R1 and R2, carries out of bit position 40 of the updated address are ignored, and the contents of bit positions 32-39 of general registers R1 and R2 are set to zeros. In the 31-bit addressing mode, the contents of bit positions 33-63 of general registers R1 and R2 constitute the addresses of the first and second operands, respectively, and the contents of bit positions 0-32 are ignored; bits 33-63 of the updated addresses replace the corresponding bits in general registers R1 and R2, carries out of bit position 33 of the updated address are ignored, and the content of bit position 32 of general registers R1 and R2 is set to zero. In the 64-bit addressing mode, the contents of bit positions 0-63 of general registers R1 and R2 constitute the addresses of the first and second operands, respectively; bits 0-63 of the updated addresses replace the contents of general registers R1 and R2, and carries out of bit position 0 are ignored.

In both the 24-bit and the 31-bit addressing modes, the contents of bit positions 32-63 of general register R2 + 1 form a 32-bit unsigned binary integer which specifies the number of bytes in the first and second operands, and the contents of bit positions 0-31 are ignored; bits 32-63 of the updated value replace the corresponding bits in general register R2 + 1. In the 64-bit addressing mode, the contents of bit positions 0-63 of general register R2 + 1 form a 64-bit unsigned binary integer which specifies the number of bytes in the first and second operands; and the updated value replaces the contents of general register R2 + 1.

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers R1, R2, and R2 + 1, always remain unchanged. Fig. 5 shows the contents of the general registers just described.

In the access-register mode, access registers 1, R1, and R2 specify the address spaces containing the parameter block, first, and second operands, respectively.

The result is obtained as if processing starts at the left end of both the first and second operands and proceeds to the right, block by block. The operation is ended when the number of bytes in the second operand as specified in general register R2 + 1 have been processed and placed at the first-operand location (called normal completion) or when a CPU-determined number of blocks that is less than the length of the second operand have been processed (called partial completion). The CPU-determined number of blocks depends on the model, and may be a different number each time the instruction is executed. The CPU-determined number of blocks is usually nonzero. In certain unusual situations, this number may be zero, and condition code 3 may be set with no progress. However, the CPU protects against endless reoccurrence of this no-progress case.

The results in the first-operand location and the chaining-value field are unpredictable if any of the following situations occur:
1. The cryptographic-key field overlaps any portion of the first operand.
2. The chaining-value field overlaps any portion of the first operand or the second operand.
3. The first and second operands overlap destructively. Operands are said to overlap destructively when the first-operand location would be used as a source after data would have been moved into it, assuming processing to be performed from left to right and one byte at a time.

When the operation ends due to normal completion, condition code 0 is set and the resulting value in R2 + 1 is zero. When the operation ends due to partial completion, condition code 3 is set and the resulting value in R2 + 1 is nonzero.

When a storage-alteration PER event is recognized, fewer than 4K additional bytes are stored into the first-operand locations before the event is reported.

When the second-operand length is initially zero, the parameter block, first, and second operands are not accessed, general registers R1, R2, and R2 + 1 are not changed, and condition code 0 is set.

When the contents of the R1 and R2 fields are the same, the contents of the designated registers are incremented only by the number of bytes processed, not by twice the number of bytes processed.

As observed by other CPUs and channel programs, references to the parameter block and storage operands may be multiple-access references, accesses to these storage locations are not necessarily block-concurrent, and the sequence of these accesses or references is undefined.

In certain unusual situations, instruction execution may complete by setting condition code 3 without updating the registers and chaining value to reflect the last unit of the first and second operands processed. The size of the unit processed in this case depends on the situation and the model, but is limited such that the portion of the first and second operands which have been processed and not reported do not overlap in storage. In all cases, change bits are set and PER storage-alteration events are reported, when applicable, for all first-operand locations processed.

Access exceptions may be reported for a larger portion of an operand than is processed in a single execution of the instruction; however, access exceptions are not recognized for locations beyond the length of an operand nor for locations more than 4K bytes beyond the current location being processed.

Symbols Used in Function Descriptions

The following symbols are used in the subsequent description of the CIPHER MESSAGE and CIPHER MESSAGE WITH CHAINING functions. For

data-encryption-algorithm (DEA) functions, the DEA-key-parity bit in each byte of the DEA key is ignored, and the operation proceeds normally, regardless of the DEA-key parity of the key. Further description of the data-encryption algorithm may be found in *Data Encryption Algorithm*, ANSI-X3.92.1981, American National Standard for Information Systems.

Fig. 6 illustrates the symbol for the Bit-Wise Exclusive Or, Fig. 7 illustrates the symbols for DEA Encryption and Decryption.

KM-Query (KM Function Code 0)

The locations of the operands and addresses used by the instruction are as shown in Fig. 5.

The parameter block used for the function KM-Query has the format shown in Fig. 8

A 128-bit status word is stored in the parameter block. Bits 0-127 of this field correspond to function codes 0-127, respectively, of the CIPHER MESSAGE instruction. When a bit is one, the corresponding function is installed; otherwise, the function is not installed.

Condition code 0 is set when execution of the KM-Query function completes; condition code 3 is not applicable to this function.

KM-DEA (KM Function Code 1)

The locations of the operands and addresses used by the instruction are as shown in Fig. 5.

The parameter block used for the function KM-DEA has the format shown in Fig. 9.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 8-byte plaintext blocks (P1, P2, ..., Pn) in operand 2 are enciphered using the DEA algorithm with the 64-bit cryptographic key in the parameter block. Each plaintext block is independently enciphered; that is, the encipher operation is performed without chaining. The ciphertext blocks (C1, C2, ..., Cn) are stored in operand 1. The operation is shown in Fig. 10.

When the modifier bit in general register 0 is one, a decipher operation is performed. The 8-byte ciphertext blocks (C1, C2, ..., Cn) in operand 2 are deciphered using the DEA algorithm with the 64-bit cryptographic key in the parameter block. Each ciphertext block is independently deciphered; that is, the decipher operation is performed without chaining. The plaintext blocks (P1, P2, ..., Pn) are stored in operand 1. The KM-DEA decipher operation is shown in Fig. 11.

KM-TDEA-128 (KM Function Code 2)

The locations of the operands and addresses used by the instruction are as shown in Fig. 5. The parameter block used for the KM-TDEA-128 function is shown in Fig. 12.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 8-byte plaintext blocks (P1, P2, ..., Pn) in operand 2 are enciphered using the TDEA (triple DEA) algorithm with the two 64-bit cryptographic keys in the parameter block. Each plaintext block is independently enciphered; that is, the encipher operation is performed without chaining. The ciphertext blocks (C1, C2, ..., Cn) are stored in operand 1. The KM-TDEA-128 encipher operation is shown in Fig. 13.

When the modifier bit in general register 0 is one, a decipher operation is performed. The 8-byte ciphertext blocks (C1, C2, ..., Cn) in operand 2 are deciphered using the TDEA algorithm with the two 64-bit cryptographic keys in the parameter block. Each ciphertext block is independently deciphered; that is, the decipher operation is performed without chaining. The plaintext blocks (P1, P2, ..., Pn) are stored in operand 1. The KM-TDEA-128 decipher operation is shown in Fig. 14.

KM-TDEA-192 (KM Function Code 3)

The locations of the operands and addresses used by the instruction are as shown in Fig. 5.

The parameter block used for the KM-TDEA-192 function has the format shown in Fig. 15.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 8-byte plaintext blocks (P1, P2, ..., Pn) in operand 2 are enciphered using the TDEA algorithm with the three 64-bit cryptographic keys in the parameter block. Each plaintext block is

independently enciphered; that is, the encipher operation is performed without chaining. The ciphertext blocks (C1, C2, ..., Cn) are stored in operand 1. The KM-TDEA-192 encipher operation is shown in Fig. 16.

When the modifier bit in general register 0 is one, a decipher operation is performed. The 8-byte ciphertext blocks (C1, C2, ..., Cn) in operand 2 are deciphered using the TDEA algorithm with the three 64-bit cryptographic keys in the parameter block. Each ciphertext block is independently deciphered; that is, the decipher operation is performed without chaining. The plaintext blocks (P1, P2, ..., Pn) are stored in operand 1. The KM-TDEA-192 decipher operation is shown in Fig. 17.

KMC-Query (KMC Function Code 0)

The locations of the operands and addresses used by the instruction are as shown in Fig. 5.

The parameter block used for the KMC-Query function has the format shown in Fig. 18.

A 128-bit status word is stored in the parameter block. Bits 0-127 of this field correspond to function codes 0-127, respectively, of the CIPHER MESSAGE WITH CHAINING instruction. When a bit is one, the corresponding function is installed; otherwise, the function is not installed.

Condition code 0 is set when execution of the KMC-Query function completes; condition code 3 is not applicable to this function.

KMC-DEA (KMC Function Code 1)

The locations of the operands and addresses used by the instruction are as shown in Fig. 5.

The parameter block used for the KMC-DEA function has the format shown in Fig. 19.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 8-byte plaintext blocks (P1, P2, ..., Pn) in operand 2 are enciphered using the DEA algorithm with the 64-bit cryptographic key and the 64-bit chaining value in the parameter block.

The chaining value, called the initial chaining value (ICV), for deriving the first ciphertext block is the chaining value in the parameter block; the chaining value for deriving each subsequent ciphertext block is the corresponding previous ciphertext block. The ciphertext blocks (C1, C2, ..., Cn) are stored in operand 1. The last ciphertext block is the output chaining value (OCV) and is stored into the chaining-value field of the parameter block. The KMC-DEA encipher operation is shown in Fig. 20.

When the modifier bit in general register 0 is one, a decipher operation is performed. The 8-byte ciphertext blocks (C1, C2, ..., Cn) in operand 2 are deciphered using the DEA algorithm with the 64-bit cryptographic key and the 64-bit chaining value in the parameter block.

The chaining value, called the initial chaining value (ICV), for deriving the first plaintext block is in the parameter block; the chaining value for deriving each subsequent plaintext block is the corresponding previous ciphertext block. The plaintext blocks (P1, P2, ..., Pn) are stored in operand 1. The last ciphertext block is the output chaining value (OCV) and is stored into the chaining-value field in the parameter block. The KMC-DEA decipher operation is shown in Fig. 21.

KMC-TDEA-128 (KMC Function Code 2)

The locations of the operands and addresses used by the instruction are as shown in Fig. 5.

The parameter block used for the KMC-TDEA-128 function has the format shown in Fig. 22.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 8-byte plaintext blocks (P1, P2, ..., Pn) in operand 2 are enciphered using the TDEA algorithm with the two 64-bit cryptographic keys and the 64-bit chaining value in the parameter block.

The chaining value, called the initial chaining value (ICV), for deriving the first ciphertext block is the chaining value in the parameter block; the chaining value for deriving each subsequent ciphertext block is the corresponding previous ciphertext block. The ciphertext blocks (C1, C2, ..., Cn) are stored in operand 1. The last ciphertext block is the output chaining value (OCV) and is stored into the chaining-value field of the parameter block. The KMC-TDEA-128 encipher operation is shown in Fig. 23.

When the modifier bit in general register 0 is one, a decipher operation is performed. The 8-byte ciphertext blocks (C1, C2, ..., Cn) in operand 2 are deciphered using the TDEA algorithm with the two 64-bit cryptographic keys and the 64-bit chaining value in the parameter block.

The chaining value, called the initial chaining value (ICV), for deriving the first plaintext block is in the parameter block; the chaining value for deriving each subsequent plaintext block is the corresponding previous ciphertext block. The plaintext blocks (P1, P2, ..., Pn) are stored in operand 1. The last ciphertext block is the output chaining value (OCV) and is stored into the chaining-value field in the parameter block. The KMC-TDEA-128 operation is shown in Fig. 24.

KMC-TDEA-192 (KMC Function Code 3)

The locations of the operands and addresses used by the instruction are as shown in Fig. 5.

The parameter block used for the KMC-TDEA-192 function has the format shown in Fig. 25.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 8-byte plaintext blocks (P1, P2, ..., Pn) in operand 2 are enciphered using the TDEA algorithm with the three 64-bit cryptographic keys and the 64-bit chaining value in the parameter block.

The chaining value, called the initial chaining value (ICV), for deriving the first ciphertext block is the chaining value in the parameter block; the chaining value for deriving each subsequent ciphertext block is the corresponding previous ciphertext block. The ciphertext blocks (C1, C2, ..., Cn) are stored in operand 1. The last ciphertext block is the output chaining value (OCV) and is stored into the chaining-value field of the parameter block. The KMC-TDEA-192 encipher operation is shown in Fig. 26.

When the modifier bit in general register 0 is one, a decipher operation is performed. The 8-byte ciphertext blocks (C1, C2, ..., Cn) in operand 2 are deciphered using the TDEA algorithm with the three 64-bit cryptographic keys and the 64-bit chaining value in the parameter block.

The chaining value, called the initial chaining value (ICV), for deriving the first plaintext block is in the parameter block; the chaining value for deriving each subsequent plaintext block is the corresponding previous

ciphertext block. The plaintext blocks (P1, P2, ..., Pn) are stored in operand 1. The last ciphertext block is the output chaining value (OCV) and is stored into the chaining-value field in the parameter block. The KMC-TDEA-192 decipher operation is shown in Fig. 27.

Special Conditions for KM and KMC

A specification exception is recognized and no other action is taken if any of the following occurs:

1. Bits 57-63 of general register 0 specify an unassigned or uninstalled function code.
2. The R1 or R2 field designates an odd-numbered register or general register 0.
3. The second operand length is not a multiple of the data block size of the designated function (see Figure 7-3 on page 7-35 to determine the data block sizes for CIPHER MESSAGE functions; see Figure 7-4 on page 7-35 to determine the data block sizes for CIPHER MESSAGE WITH CHAINING functions). This specification-exception condition does not apply to the query functions.

Resulting Condition Code:
0 Normal completion
1 --
2 --
3 Partial completion

Program Exceptions:
Access (fetch, operand 2 and cryptographic key; store, operand 1; fetch and store, chaining value)
Operation (if the message-security assist is not installed)
Specification

Fig. 28 is a table showing the priority of execution of KM and KMC.

Programming Notes:
1. When condition code 3 is set, the general registers containing the operand addresses and length, and, for CIPHER MESSAGE WITH CHAINING, the chaining value in the parameter block, are usually updated such that the program can simply branch back to the instruction to continue the operation.

For unusual situations, the CPU protects against endless reoccurrence of the no- progress case and also protects against setting condition code 3 when the portion of the first and second operands to be reprocessed overlap in storage. Thus, the program can safely branch back to the instruction whenever condition code 3 is set with no exposure to an endless loop and no exposure to incorrectly retrying the instruction.

2. If the length of the second operand is nonzero initially and condition code 0 is set, the registers are updated in the same manner as for condition code 3. For CIPHER MESSAGE WITH CHAINING, the chaining value in this case is such that additional operands can be processed as if they were part of the same chain.
3. To save storage, the first and second operands may overlap exactly or the starting point of the first operand may be to the left of the starting point of the second operand. In either case, the overlap is not destructive.

CRYPTO COPROCESSOR:

The preferred embodiment provides a crypto coprocessor which can be used with the instructions described herein and to execute cipher messages and assist in a variety of chaining message tasks which can be employed for chained and cryptographic use with the appropriate instructions.

Fig. 29 illustrates our cryptographic coprocessor which is directly attached to a data path common to all internal execution units on the general purpose microprocessor, which has multiple execution pipelines. The microprocessor internal bus (1) is common to all other execution units is attached to the cryptographic control unit (2), and the control unit watches the bus for processor instructions that it should execute.

The cryptographic control unit provides a cryptographic coprocessor directly attached to a data path common to all internal execution units of the central processing unit on a general purpose microprocessor providing the available hardware ($E_0 \ldots E_n$), or from a combination thereof in the preferred embodiment having multiple execution pipelines) for the central processing unit. When a cryptographic instruction is encountered in the command register (3), the control unit (2) invokes the appropriate algorithm from the available hardware. Operand data is delivered over the same internal microprocessor bus via an input FIFO register (4). When an operation is completed the a flag is set in a status register (6) and the results are available to be read out from the output FIFO register (5).

The illustrated preferred embodiment of our invention is designed to be extensible to include as many hardware engines as required by a particular implementation depending on the performance goals of the system. The data paths to the input and output registers (7) are common among all engines.

The preferred embodiment of the invention cryptographic functions are implemented in execution unit hardware on the CPU and this implementation enables a lower latency for calling and executing encryption operations and increases the efficiency.

This decreased latency greatly enhances the capability of general purpose processors in systems that frequently do many encryption operations, particularly when only small amounts of data are involved. This allows an implementation that can significantly accelerate the processes involved in doing secure online transactions. The most common methods of securing online transactions involve a set of three algorithms. The first algorithm is only used one time in a session, and may be implemented in hardware or software, while the other operations are invoked with every transaction of the session, and the cost in latency of calling external hardware as well as the cost in time to execute the algorithm in software are both eliminated with this invention.

In Fig. 30 we have shown conceptually how to implement what we have in a preferred embodiment implemented in a mainframe computer having the microprocessor described above which can effectively be used, as we have experimentally proven within IBM, in a commercial implementation of the long displacement facility computer architected instruction format the instructions are used by programmers, usually today "C" programmers. These instruction formats stored in the storage medium may be executed natively in a Z/Architecture IBM Server, or alternatively in machines executing other architectures. They can be emulated in the existing and in future IBM mainframe servers and on other machines of IBM (e.g. pSeries Servers and xSeries Servers). They can be executed in machines running Linux on a wide variety of machines using hardware manufactured by IBM, Intel, AMD, Sun Microsystems and others. Besides execution on that hardware under a Z/Architecture, Linux can be used as well as machines which use emulation by Hercules, UMX, FXI or Platform Solutions, where generally execution is in an emulation mode. In emulation mode the specific instruction being emulated is decoded, and a subroutine built to implement the individual instruction, as in a "C" subroutine or driver, or some other method of providing a driver for the specific hardware as is within the skill of those in the art after understanding the description of the preferred

embodiment. Various software and hardware emulation patents including, but not limited to US 5551013, US6009261, US5574873, US6308255, US6463582 and US5790825, illustrate the variety of known ways to achieve emulation of an instruction format architected for a different machine for a target machine available to those skilled in the art, as well as those commercial software techniques used by those referenced above.

In the preferred embodiment the existing pre-long displacement instruction formats for a non superscalar instruction form the operand storage address by the summing of the base register and 12 bit unsigned displacement or the base register, the index register, and the 12 bit unsigned displacement and the new long displacement instruction formats form the operand storage address by the summing of the base register and the 20 bit signed displacement or the base register, the index register, and the 20 bit signed displacement.

As illustrated by Fig. 30, these instructions are executed in hardware by a processor or by emulation of said instruction set by software executing on a computer having a different native instruction set.

In Fig. 30, #501 shows a computer memory storage containing instructions and data. The long displacement instructions described in this invention would initially stored in this computer. #502 shows a mechanism for fetching instructions from a computer memory and may also contain local buffering of these instructions it has fetched. Then the raw instructions are transferred to an instruction decoder, #503, where it determines what type of instruction has been fetched. #504, shows a mechanism for executing instructions. This may include loading data into a register from memory, #501, storing data back to memory from a register, or performing some type of arithmetic or logical operation. This exact type of operation to be performed has been previously determined by the instruction decoder. The long displacement instructions described in this invention would be executed here. If the long displacement instructions are being executed natively on a computer system, then this diagram is complete as described above. However, if an instruction set architecture, containing long displacement instructions, is being emulated on another computer, the above process would be implemented in software on a host computer, #505. In this case, the above stated mechanisms would typically be implemented as one or more software subroutines within the emulator software. In both cases an instruction is fetched, decoded and executed.

More particularly, these architected instructions can be used with a computer architecture with existing instruction formats with a 12 bit unsigned displacement used to form the operand storage address and also one having additional instruction formats that provide a additional displacement bits, preferably 20 bits, which comprise an extended signed displacement used to form the operand storage address. These computer architected instructions comprise computer software, stored in a computer storage medium, for producing the code running of the processor utilizing the computer software, and comprising the instruction code for use by a compiler or emulator/interpreter which is stored in a computer storage medium 501, and wherein the first part of the instruction code comprises an operation code which specified the operation to be performed and a second part which designates the operands for that participate. The long displacement instructions permit additional addresses to be directly addressed with the use of the long displacement facility instruction.

As illustrated by Fig. 30, these instructions are executed in hardware by a processor or by emulation of said instruction set by software executing on a computer having a different native instruction set.

In accordance with the computer architecture of the preferred embodiment the displacement field is defined as being in two parts, the least significant part being 12 bits called the DL, DL1 for operand 1 or DL2 for operand 2, and the most significant part being 8 bits called the DH, DH1 for operand 1 or DH2 for operand 2.

Furthermore, the preferred computer architecture has an instruction format such that the opcode is in bit positions 0 through 7 and 40 through 47, a target register called R1 in bit positions 8 through 11, an index register called X2 in bit positions 12 through 15, a base register called B2 in bit positions 16 through 19, a displacement composed of two parts with the first part called DL2 in bit positions 20 through 31 and the second part called DH2 in bit positions 32 through 39.

This computer architecture has an instruction format such that the opcode is in bit positions 0 through 7 and 40 through 47, a target register called R1 in bit positions 8 through 11, an source register called R3 in bit positions 12 through 15, a base register called B2 in bit positions 16 through 19, a displacement composed of two parts with the first part called DL2 in bit positions 20 through 31 and the second part called DH2 in bit positions 32 through 39.

Furthermore, our computer architecture instructions having a long displacement facility has an instruction format such that the opcode is in bit positions 0 through 7 and 40 through 47, a target register called R1 in bit positions 8 through 11, a mask value called M3 in bit positions 12 through 15, a base register called B2 in bit positions 16 through 19, a displacement composed of two parts with the first part called DL2 in bit positions 20 through 31 and the second part called DH2 in bit positions 32 through 39.

As illustrated, our preferred computer architecture with its long displacement facility has an instruction format such that the opcode is in bit positions 0 through 7 and 40 through 47, an immediate value called I2 in bit positions 8 through 15, a base register called B2 in bit positions 16 through 19, a displacement composed of two parts with the first part called DL1 in bit positions 20 through 31 and the second part called DH1 in bit positions 32 through 39.

Our long displacement facility computer architecture operates effectively when using new instructions which are created that only use the instruction format with the new 20 bit unsigned displacement.

A specific embodiment of our computer architecture utilizes existing instructions which have the instruction formats that only have the 12 bit unsigned displacement and are now defined to be in the new instruction formats to have either the existing 12 bit unsigned displacement value when the high order 8 bits of the displacement, field DH, are all zero, or a 20 bit signed value when the high order 8 bits of the displacement, field DH, is non-zero.

An apparatus for enciphering or deciphering storage of a computing environment, said apparatus comprising:

means for specifying, via an instruction, a unit of storage to be enciphered or deciphered; and

means for enciphering or deciphering the unit of storage.

CLAIMS

1. A method of enciphering or deciphering storage of a computing environment, said method comprising:

specifying, via an instruction, a unit of storage to be enciphered or deciphered; and

enciphering or deciphering the unit of storage.

2. The method of claim 1 wherein the specifying comprises providing location information of a data structure associated with the unit of storage.

3. The method of claim 2 wherein the location information comprises an origin of the data structure.

4. The method of claim 3 wherein the location information further comprises an index of an entry of the data structure, said entry corresponding to the unit of storage.

5. The method of claim 1, wherein the enciphering or deciphering comprises providing a cryptographic key by said instruction.

6. The method of claim 1 wherein the unit of storage comprises one of a segment of storage and a region of storage, and wherein said data structure comprises one of a segment data structure and a region data structure.

7. The method of claim 1 wherein the enciphering or deciphering is performed via the instruction.

8. The method of claim 1 wherein the enciphering or deciphering comprises setting a cryptographic key associated with the unit of storage.

9. The method of claim 1 wherein the data structure comprises a plurality of entries, and wherein the enciphering or deciphering further comprises employing an index to obtain the entry associated with the unit of storage to be enciphered or deciphered, said entry having the cryptographic key.

10. The method of claim 1 wherein the unit of storage comprises a segment of storage, said segment of storage comprising a plurality of pages of storage.

11. The method of claim 1 wherein the unit of storage comprises a region of storage, said region of storage comprising a plurality of segments of storage, a segment of storage comprising a plurality of pages of storage.

12. The method of claim 1 wherein the specifying comprises specifying a plurality of units of storage, and the enciphering or deciphering comprises enciphering or deciphering the plurality of units of storage.

13. The method of claim 12 wherein the enciphering or deciphering comprises a chaining operation for enciphering or deciphering the plurality of units of storage.

14. The method of claim 1 wherein the storage comprises virtual storage.

15. The method of claim 1 wherein the instruction is implemented in as least one of hardware, firmware and software.

16. The method of claim 1 wherein the instruction is executed by a processing unit emulating an architecture of the instruction, said architecture of the instruction being different than an architecture of the processing unit.

17. A system for enciphering or deciphering storage of a computing environment, said system comprising means for carrying out the steps of any preceding method claim.

18. A computer program product comprising:

at least one computer usable medium having computer readable program code for performing the method of any preceding method claim.