

INSTRUCTIONS FOR PERFORMING AN OPERATION ON A OPERAND IN MEMORY AND
SUBSEQUENTLY LOADING AN ORIGINAL VALUE OF SAID
OPERAND IN A REGISTER

FIELD OF THE INVENTION

The present invention is related to computer systems and more particularly to computer system processor instruction functionality.

BACKGROUND

Trademarks: IBM® is a registered trademark of International Business Machines Corporation, Armonk, New York, U.S.A. S/390, Z900, z990 and z10 and other product names may be registered trademarks or product names of International Business Machines Corporation or other companies.

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 Eighth Edition of the IBM® z/Architecture® Principles of Operation which was published February, 2009 has become the standard published reference as SA22-7832-07 and is incorporated in IBM’s z10® mainframe servers including the IBM System z10® Enterprise Class servers.

Referring to FIG. 1A, representative components of a Host Computer system 50 are portrayed. Other arrangements of components may also be employed in a computer system, which are well known in the art. The representative Host Computer 50 comprises one or more CPUs 1 in communication with main store (Computer Memory 2) as well as I/O

interfaces to storage devices 11 and networks 10 for communicating with other computers or SANs and the like. The CPU 1 is compliant with an architecture having an architected instruction set and architected functionality. The CPU 1 may have Dynamic Address Translation (DAT) 3 for transforming program addresses (virtual addresses) into real address of memory. A DAT typically includes a Translation Lookaside Buffer (TLB) 7 for caching translations so that later accesses to the block of computer memory 2 do not require the delay of address translation. Typically a cache 9 is employed between Computer Memory 2 and the Processor 1. The cache 9 may be hierarchical having a large cache available to more than one CPU and smaller, faster (lower level) caches between the large cache and each CPU. In some implementations the lower level caches are split to provide separate low level caches for instruction fetching and data accesses. In an embodiment, an instruction is fetched from memory 2 by an instruction fetch unit 4 via a cache 9. The instruction is decoded in an instruction decode unit (6) and dispatched (with other instructions in some embodiments) to instruction execution units 8. Typically several execution units 8 are employed, for example an arithmetic execution unit, a floating point execution unit and a branch instruction execution unit. The instruction is executed by the execution unit, accessing operands from instruction specified registers or memory as needed. If an operand is to be accessed (loaded or stored) from memory 2, a load store unit 5 typically handles the access under control of the instruction being executed. Instructions may be executed in hardware circuits or in internal microcode (firmware) or by a combination of both.

In FIG. 1B, an example of an emulated Host Computer system 21 is provided that emulates a Host computer system 50 of a Host architecture. In the emulated Host Computer system 21, the Host processor (CPU) 1 is an emulated Host processor (or virtual Host processor) and comprises an emulation processor 27 having a different native instruction set architecture than that of the processor 1 of the Host Computer 50. The emulated Host Computer system 21 has memory 22 accessible to the emulation processor 27. In the example embodiment, the Memory 27 is partitioned into a Host Computer Memory 2 portion and an Emulation Routines 23 portion. The Host Computer Memory 2 is available to programs of the emulated Host Computer 21 according to Host Computer Architecture. The emulation Processor 27 executes native instructions of an architected instruction set of an architecture other than that of the emulated processor 1, the native instructions obtained from Emulation Routines

memory 23, and may access a Host instruction for execution from a program in Host Computer Memory 2 by employing one or more instruction(s) obtained in a Sequence & Access/Decode routine which may decode the Host instruction(s) accessed to determine a native instruction execution routine for emulating the function of the Host instruction accessed. Other facilities that are defined for the Host Computer System 50 architecture may be emulated by Architected Facilities Routines, including such facilities as General Purpose Registers, Control Registers, Dynamic Address Translation and I/O Subsystem support and processor cache for example. The Emulation Routines may also take advantage of function available in the emulation Processor 27 (such as general registers and dynamic translation of virtual addresses) to improve performance of the Emulation Routines. Special Hardware and Off-Load Engines may also be provided to assist the processor 27 in emulating the function of the Host Computer 50.

In a mainframe, architected machine instructions are used by programmers, usually today “C” programmers often by way of a compiler application. These instructions 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 as described at <http://www.turbohercules.com>, <http://www.hercules-390.org> and <http://www.funsoft.com>. In emulation mode, emulation software is executed by a native processor to emulate the architecture of an emulated processor.

The native processor 27 typically executes emulation software 23 comprising either firmware or a native operating system to perform emulation of the emulated processor. The emulation software 23 is responsible for fetching and executing instructions of the emulated processor architecture. The emulation software 23 maintains an emulated program counter to keep track of instruction boundaries. The emulation software 23 may fetch one or more emulated machine instructions at a time and convert the one or more emulated machine

instructions to a corresponding group of native machine instructions for execution by the native processor 27. These converted instructions may be cached such that a faster conversion can be accomplished. Notwithstanding, the emulation software must maintain the architecture rules of the emulated processor architecture so as to assure operating systems and applications written for the emulated processor operate correctly. Furthermore the emulation software must provide resources identified by the emulated processor 1 architecture including, but not limited to control registers, general purpose registers, floating point registers, dynamic address translation function including segment tables and page tables for example, interrupt mechanisms, context switch mechanisms, Time of Day (TOD) clocks and architected interfaces to I/O subsystems such that an operating system or an application program designed to run on the emulated processor, can be run on the native processor having the emulation software.

A specific instruction being emulated is decoded, and a subroutine called to perform the function of the individual instruction. An emulation software function 23 emulating a function of an emulated processor 1 is implemented, for example, in a "C" subroutine or driver, or some other method of providing a driver for the specific hardware as will be 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 for a "Multiprocessor for hardware emulation" of Beausoleil et al., and US6009261: Preprocessing of stored target routines for emulating incompatible instructions on a target processor" of Scalzi et al; and US5574873: Decoding guest instruction to directly access emulation routines that emulate the guest instructions, of Davidian et al; US6308255: Symmetrical multiprocessing bus and chipset used for coprocessor support allowing non-native code to run in a system, of Gorishek et al; and US6463582: Dynamic optimizing object code translator for architecture emulation and dynamic optimizing object code translation method of Lethin et al; and US5790825: Method for emulating guest instructions on a host computer through dynamic recompilation of host instructions of Eric Traut. These references illustrate a 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 US Patent No. 7,627,723 B1, issued December 1, 2009, Buck et al., “Atomic Memory Operators in a Parallel Processor,” methods, apparatuses, and systems are presented for updating data in memory while executing multiple threads of instructions, involving receiving a single instruction from one of a plurality of concurrently executing threads of instructions, in response to the single instruction received, reading data from a specific memory location, performing an operation involving the data read from the memory location to generate a result, and storing the result to the specific memory location, without requiring separate load and store instructions, and in response to the single instruction received, precluding another one of the plurality of threads of instructions from altering data at the specific memory location while reading of the data from the specific memory location, performing the operation involving the data, and storing the result to the specific memory location.

U.S. Patent No. 5,838,960, issued November 17, 1998, Harriman, Jr., “Apparatus for Performing an Atomic Add Instructions,” describes a pipeline processor having an add circuit configured to execute separate atomic add instructions in consecutive clock cycles, wherein each separate atomic add instructions can be updating the same memory address location. In one embodiment, the add circuit includes a carry-save-add circuit coupled to a set of carry propagate adder circuits. The carry-save-add circuit is configured to perform an add operation in one processor clock cycle and the set of carry propagate adder circuits are configured to propagate, in subsequent clock cycles, a carry generated by the carry-save-add circuit. The add circuit is further configured to feedforward partially propagated sums to the carry-save-add circuit as at least one operand for subsequent atomic add instructions. In one embodiment, the pipeline processor is implemented on a multitasking computer system architecture supporting multiple independent processors dedicated to processing data packets.

What is needed is new instruction functionality consistent with existing architecture that relieves dependency on architecture resources such as general registers, improves functionality and performance of software versions employing the new instruction.

SUMMARY

In an embodiment, an arithmetic/logical instruction is executed, wherein the instruction comprises an interlocked memory operand, the arithmetic/logical instruction comprising an opcode field, a first register field specifying a first operand in a first register, a second register field specifying a second register the second register specifying location of a second operand in memory, and a third register field specifying a third register, the execution of the arithmetic/logical instruction comprises: obtaining by a processor, a second operand from a location in memory specified by the second register, the second operand consisting of a value; obtaining a third operand from the third register; performing an opcode defined arithmetic operation or a logical operation based on the obtained second operand and the obtained third operand to produce a result; storing the produced result in the location in memory; and saving the value of the obtained second operand in the first register, wherein the value is not changed by executing the instruction.

In an embodiment, a condition code is saved, the condition code indicating the result is zero or the result is not zero.

In an embodiment, the opcode defined arithmetic operation is an arithmetic or logical ADD, and the opcode defined logical operation is any one of an AND, an EXCLUSIVE-OR, or an OR, and the execution comprises: responsive to the result of the logical operation being negative, saving the condition code indicating the result is negative; responsive to the result of the logical operation being positive, saving the condition code indicating the result is positive; and responsive to the result of the logical operation being an overflow, saving the condition code indicating the result is an overflow.

In an embodiment, operand size is specified by the opcode, wherein one or more first opcodes specify 32 bit operands and one or more second opcodes specify 64 bit operands.

In an embodiment, the arithmetic/logical instruction further comprises the opcode consisting of two separate opcode fields, a first displacement field and a second displacement field, wherein the location in memory is determined by adding contents of the second register to a

signed displacement value, the signed displacement value comprising a sign extended value of the first displacement field concatenated to the second displacement field.

In an embodiment, the execution further comprises: responsive to the opcode being a first opcode and the second operand not being on a 32 bit boundary, generating a specification exception; and responsive to the opcode being a second opcode and the second operand not being on a 64 bit boundary, generating a specification exception.

In an embodiment, the processor is a processor in a multi-processor system, and the execution further comprises: the obtaining the second operand comprising preventing other processors of the multi-processor system from accessing the location in memory between said obtaining of the second operand and storing a result at the second location in memory; and upon said storing the produced result, permitting other processors of the multi-processor system to access the location in memory.

The above as well as additional objectives, features, and advantages embodiments will become apparent in the following written description.

BRIEF DESCRIPTION OF THE DRAWINGS

Embodiments of the invention will now be described, by way of example only, with reference to the accompanying drawings in which:

FIG. 1A is a diagram depicting an example Host computer system;

FIG. 1B is a diagram depicting an example emulation Host computer system;

FIG. 1C is a diagram depicting an example computer system;

FIG. 2 is a diagram depicting an example computer network;

FIG. 3 is a diagram depicting an elements of a computer system;

FIGs. 4A-4C depict detailed elements of a computer system;

FIGs. 5A-5F depict machine instruction format of a computer system;

FIGs. 6A-6B depict an example flow of an embodiment; and

FIG. 7 depicts an example context switch flow.

DETAILED DESCRIPTION

An embodiment may be practiced by software (sometimes referred to Licensed Internal Code, Firmware, Micro-code, Milli-code, Pico-code and the like, any of which would be consistent with the embodiments). Referring to FIG. 1A, software program code is typically accessed by the processor also known as a CPU (Central Processing Unit) 1 of the system 50 from long-term storage media 7, such as a CD-ROM drive, tape drive or hard drive. The software program code may be embodied on any of a variety of known media for use with a data processing system, such as a diskette, hard drive, or CD-ROM. The code may be distributed on such media, or may be distributed to users from the computer memory 2 or storage of one computer system over a network 10 to other computer systems for use by users of such other systems.

Alternatively, the program code may be embodied in the memory 2, and accessed by the processor 1 using the processor bus. Such program code includes an operating system which controls the function and interaction of the various computer components and one or more application programs. Program code is normally paged from dense storage media 11 to high-speed memory 2 where it is available for processing by the processor 1. The techniques and methods for embodying software program code in memory, on physical media, and/or distributing software code via networks are well known and will not be further discussed herein. Program code, when created and stored on a tangible medium (including but not limited to electronic memory modules (RAM), flash memory, Compact Discs (CDs), DVDs, Magnetic Tape and the like is often referred to as a "computer program product". The computer program product medium is typically readable by a processing circuit preferably in a computer system for execution by the processing circuit.

FIG. 1C illustrates a representative workstation or server hardware system. The system 100 of FIG. 1C comprises a representative computer system 101, such as a personal computer, a workstation or a server, including optional peripheral devices. The workstation 101 includes one or more processors 106 and a bus employed to connect and enable communication between the processor(s) 106 and the other components of the system 101 in accordance with known techniques. The bus connects the processor 106 to memory 105 and long-term

storage 107 which can include a hard drive (including any of magnetic media, CD, DVD and Flash Memory for example) or a tape drive for example. The system 101 might also include a user interface adapter, which connects the microprocessor 106 via the bus to one or more interface devices, such as a keyboard 104, mouse 103, a Printer/scanner 110 and/or other interface devices, which can be any user interface device, such as a touch sensitive screen, digitized entry pad, etc. The bus also connects a display device 102, such as an LCD screen or monitor, to the microprocessor 106 via a display adapter.

The system 101 may communicate with other computers or networks of computers by way of a network adapter capable of communicating 108 with a network 109. Example network adapters are communications channels, token ring, Ethernet or modems. Alternatively, the workstation 101 may communicate using a wireless interface, such as a CDPD (cellular digital packet data) card. The workstation 101 may be associated with such other computers in a Local Area Network (LAN) or a Wide Area Network (WAN), or the workstation 101 can be a client in a client/server arrangement with another computer, etc. All of these configurations, as well as the appropriate communications hardware and software, are known in the art.

FIG. 2 illustrates a data processing network 200 in which embodiments may be practiced. The data processing network 200 may include a plurality of individual networks, such as a wireless network and a wired network, each of which may include a plurality of individual workstations 101 201 202 203 204. Additionally, as those skilled in the art will appreciate, one or more LANs may be included, where a LAN may comprise a plurality of intelligent workstations coupled to a host processor.

Still referring to FIG. 2, the networks may also include mainframe computers or servers, such as a gateway computer (client server 206) or application server (remote server 208) which may access a data repository and may also be accessed directly from a workstation 205). A gateway computer 206 serves as a point of entry into each network 207. A gateway is needed when connecting one networking protocol to another. The gateway 206 may be preferably coupled to another network (the Internet 207 for example) by means of a communications link. The gateway 206 may also be directly coupled to one or more

workstations 101 201 202 203 204 using a communications link. The gateway computer may be implemented utilizing an IBM eServer™ zSeries® z9® Server available from IBM Corp.

Software programming code is typically accessed by the processor 106 of the system 101 from long-term storage media 107, such as a CD-ROM drive or hard drive. The software programming code may be embodied on any of a variety of known media for use with a data processing system, such as a diskette, hard drive, or CD-ROM. The code may be distributed on such media, or may be distributed to users 210 211 from the memory or storage of one computer system over a network to other computer systems for use by users of such other systems.

Alternatively, the programming code 111 may be embodied in the memory 105, and accessed by the processor 106 using the processor bus. Such programming code includes an operating system which controls the function and interaction of the various computer components and one or more application programs 112. Program code is normally paged from dense storage media 107 to high-speed memory 105 where it is available for processing by the processor 106. The techniques and methods for embodying software programming code in memory, on physical media, and/or distributing software code via networks are well known and will not be further discussed herein. Program code, when created and stored on a tangible medium (including but not limited to electronic memory modules (RAM), flash memory, Compact Discs (CDs), DVDs, Magnetic Tape and the like is often referred to as a "computer program product". The computer program product medium is typically readable by a processing circuit preferably in a computer system for execution by the processing circuit.

The cache that is most readily available to the processor (normally faster and smaller than other caches of the processor) is the lowest (L1 or level one) cache and main store (main memory) is the highest level cache (L3 if there are 3 levels). The lowest level cache is often divided into an instruction cache (I-Cache) holding machine instructions to be executed and a data cache (D-Cache) holding data operands.

Referring to FIG. 3, an exemplary processor embodiment is depicted for processor 106. Typically one or more levels of Cache 303 are employed to buffer memory blocks in order to improve processor performance. The cache 303 is a high speed buffer holding cache lines of memory data that are likely to be used. Typical cache lines are 64, 128 or 256 bytes of memory data. Separate Caches are often employed for caching instructions than for caching data. Cache coherence (synchronization of copies of lines in Memory and the Caches) is often provided by various "Snoop" algorithms well known in the art. Main storage 105 of a processor system is often referred to as a cache. In a processor system having 4 levels of cache 303 main storage 105 is sometimes referred to as the level 5 (L5) cache since it is typically faster and only holds a portion of the non-volatile storage (DASD, Tape etc) that is available to a computer system. Main storage 105 "caches" pages of data paged in and out of the main storage 105 by the Operating system.

A program counter (instruction counter) 311 keeps track of the address of the current instruction to be executed. A program counter in a z/Architecture processor is 64 bits and can be truncated to 31 or 24 bits to support prior addressing limits. A program counter is typically embodied in a PSW (program status word) of a computer such that it persists during context switching. Thus, a program in progress, having a program counter value, may be interrupted by, for example, the operating system (context switch from the program environment to the Operating system environment). The PSW of the program maintains the program counter value while the program is not active, and the program counter (in the PSW) of the operating system is used while the operating system is executing. Typically the Program counter is incremented by an amount equal to the number of bytes of the current instruction. RISC (Reduced Instruction Set Computing) instructions are typically fixed length while CISC (Complex Instruction Set Computing) instructions are typically variable length. Instructions of the IBM z/Architecture are CISC instructions having a length of 2, 4 or 6 bytes. The Program counter 311 is modified by either a context switch operation or a Branch taken operation of a Branch instruction for example. In a context switch operation, the current program counter value is saved in a Program Status Word (PSW) along with other state information about the program being executed (such as condition codes), and a new program counter value is loaded pointing to an instruction of a new program module to be executed. A branch taken operation is performed in order to permit the program to make

decisions or loop within the program by loading the result of the Branch Instruction into the Program Counter 311.

Typically an instruction Fetch Unit 305 is employed to fetch instructions on behalf of the processor 106. The fetch unit either fetches "next sequential instructions", target instructions of Branch Taken instructions, or first instructions of a program following a context switch. Modern Instruction fetch units often employ prefetch techniques to speculatively prefetch instructions based on the likelihood that the prefetched instructions might be used. For example, a fetch unit may fetch 16 bytes of instruction that includes the next sequential instruction and additional bytes of further sequential instructions.

The fetched instructions are then executed by the processor 106. In an embodiment, the fetched instruction(s) are passed to a dispatch unit 306 of the fetch unit. The dispatch unit decodes the instruction(s) and forwards information about the decoded instruction(s) to appropriate units 307 308 310. An execution unit 307 will typically receive information about decoded arithmetic instructions from the instruction fetch unit 305 and will perform arithmetic operations on operands according to the opcode of the instruction. Operands are provided to the execution unit 307 preferably either from memory 105, architected registers 309 or from an immediate field of the instruction being executed. Results of the execution, when stored, are stored either in memory 105, registers 309 or in other machine hardware (such as control registers, PSW registers and the like).

A processor 106 typically has one or more execution units 307 308 310 for executing the function of the instruction. Referring to FIG. 4A, an execution unit 307 may communicate with architected general registers 309, a decode/dispatch unit 306 a load store unit 310 and other 401 processor units by way of interfacing logic 407. An Execution unit 307 may employ several register circuits 403 404 405 to hold information that the arithmetic logic unit (ALU) 402 will operate on. The ALU performs arithmetic operations such as add, subtract, multiply and divide as well as logical function such as and, or and exclusive-or (xor), rotate and shift. Preferably the ALU supports specialized operations that are design dependent. Other circuits may provide other architected facilities 408 including condition codes and recovery support logic for example. Typically the result of an ALU operation is

held in an output register circuit 406 which can forward the result to a variety of other processing functions. There are many arrangements of processor units, the present description is only intended to provide a representative understanding of one embodiment.

An ADD instruction for example would be executed in an execution unit 307 having arithmetic and logical functionality while a Floating Point instruction for example would be executed in a Floating Point Execution having specialized Floating Point capability. Preferably, an execution unit operates on operands identified by an instruction by performing an opcode defined function on the operands. For example, an ADD instruction may be executed by an execution unit 307 on operands found in two registers 309 identified by register fields of the instruction.

The execution unit 307 performs the arithmetic addition on two operands and stores the result in a third operand where the third operand may be a third register or one of the two source registers. The Execution unit preferably utilizes an Arithmetic Logic Unit (ALU) 402 that is capable of performing a variety of logical functions such as Shift, Rotate, And, Or and XOR as well as a variety of algebraic functions including any of add, subtract, multiply, divide. Some ALUs 402 are designed for scalar operations and some for floating point. Data may be Big Endian (where the least significant byte is at the highest byte address) or Little Endian (where the least significant byte is at the lowest byte address) depending on architecture. The IBM z/Architecture is Big Endian. Signed fields may be sign and magnitude, 1's complement or 2's complement depending on architecture. A 2's complement number is advantageous in that the ALU does not need to design a subtract capability since either a negative value or a positive value in 2's complement requires only an addition within the ALU. Numbers are commonly described in shorthand, where a 12 bit field defines an address of a 4,096 byte block and is commonly described as a 4 Kbyte (Kilo-byte) block for example.

Referring to FIG. 4B, Branch instruction information for executing a branch instruction is typically sent to a branch unit 308 which often employs a branch prediction algorithm such as a branch history table 432 to predict the outcome of the branch before other conditional operations are complete. The target of the current branch instruction will be fetched and

speculatively executed before the conditional operations are complete. When the conditional operations are completed the speculatively executed branch instructions are either completed or discarded based on the conditions of the conditional operation and the speculated outcome. A typical branch instruction may test condition codes and branch to a target address if the condition codes meet the branch requirement of the branch instruction, a target address may be calculated based on several numbers including ones found in register fields or an immediate field of the instruction for example. The branch unit 308 may employ an ALU 426 having a plurality of input register circuits 427 428 429 and an output register circuit 430. The branch unit 308 may communicate with general registers 309, decode dispatch unit 306 or other circuits 425 for example.

The execution of a group of instructions can be interrupted for a variety of reasons including a context switch initiated by an operating system, a program exception or error causing a context switch, an I/O interruption signal causing a context switch or multi-threading activity of a plurality of programs (in a multi-threaded environment) for example. Preferably a context switch action saves state information about a currently executing program and then loads state information about another program being invoked. State information may be saved in hardware registers or in memory for example. State information preferably comprises a program counter value pointing to a next instruction to be executed, condition codes, memory translation information and architected register content. A context switch activity can be exercised by hardware circuits, application programs, operating system programs or firmware code (microcode, pico-code or licensed internal code (LIC) alone or in combination.

A processor accesses operands according to instruction defined methods. The instruction may provide an immediate operand using the value of a portion of the instruction, may provide one or more register fields explicitly pointing to either general purpose registers or special purpose registers (floating point registers for example). The instruction may utilize implied registers identified by an opcode field as operands. The instruction may utilize memory locations for operands. A memory location of an operand may be provided by a register, an immediate field, or a combination of registers and immediate field as exemplified by the z/Architecture long displacement facility wherein the instruction defines

a Base register, an Index register and an immediate field (displacement field) that are added together to provide the address of the operand in memory for example. Location herein typically implies a location in main memory (main storage) unless otherwise indicated.

Referring to FIG. 4C, a processor accesses storage using a Load/Store unit 310. The Load/Store unit 310 may perform a Load operation by obtaining the address of the target operand in memory 303 and loading the operand in a register 309 or another memory 303 location, or may perform a Store operation by obtaining the address of the target operand in memory 303 and storing data obtained from a register 309 or another memory 303 location in the target operand location in memory 303. The Load/Store unit 310 may be speculative and may access memory in a sequence that is out-of-order relative to instruction sequence, however the Load/Store unit 310 must maintain the appearance to programs that instructions were executed in order. A load/store unit 310 may communicate with general registers 309, decode/dispatch unit 306, Cache/Memory interface 303 or other elements 455 and comprises various register circuits, ALUs 458 and control logic 463 to calculate storage addresses and to provide pipeline sequencing to keep operations in-order. Some operations may be out of order but the Load/Store unit provides functionality to make the out of order operations to appear to the program as having been performed in order as is well known in the art.

Preferably addresses that an application program "sees" are often referred to as virtual addresses. Virtual addresses are sometimes referred to as "logical addresses" and "effective addresses". These virtual addresses are virtual in that they are redirected to physical memory location by one of a variety of Dynamic Address Translation (DAT) 312 technologies including, but not limited to simply prefixing a virtual address with an offset value, translating the virtual address via one or more translation tables, the translation tables preferably comprising at least a segment table and a page table alone or in combination, preferably, the segment table having an entry pointing to the page table. In z/Architecture, a hierarchy of translation is provided including a region first table, a region second table, a region third table, a segment table and an optional page table. The performance of the address translation is often improved by utilizing a Translation Look-aside Buffer (TLB) which comprises entries mapping a virtual address to an associated physical memory location. The entries are created when DAT 312 translates a virtual address using the

translation tables. Subsequent use of the virtual address can then utilize the entry of the fast TLB rather than the slow sequential Translation table accesses. TLB content may be managed by a variety of replacement algorithms including LRU (Least Recently used).

In the case where the Processor is a processor of a multi-processor system, each processor has responsibility to keep shared resources such as I/O, caches, TLBs and Memory interlocked for coherency. Typically "snoop" technologies will be utilized in maintaining cache coherency. In a snoop environment, each cache line may be marked as being in any one of a shared state, an exclusive state, a changed state, an invalid state and the like in order to facilitate sharing.

I/O units 304 provide the processor with means for attaching to peripheral devices including Tape, Disc, Printers, Displays, and networks for example. I/O units are often presented to the computer program by software Drivers. In Mainframes such as the z/Series from IBM, Channel Adapters and Open System Adapters are I/O units of the Mainframe that provide the communications between the operating system and peripheral devices.

The following description from the z/Architecture Principles of Operation describes an architectural view of a computer system:

STORAGE:

A computer system includes information in main storage, as well as addressing, protection, and reference and change recording. Some aspects of addressing include the format of addresses, the concept of address spaces, the various types of addresses, and the manner in which one type of address is translated to another type of address. Some of main storage includes permanently assigned storage locations. Main storage provides the system with directly addressable fast-access storage of data. Both data and programs must be loaded into main storage (from input devices) before they can be processed.

Main storage may include one or more smaller, faster-access buffer storages, sometimes called caches. A cache is typically physically associated with a CPU or an I/O processor.

The effects, except on performance, of the physical construction and use of distinct storage media are generally not observable by the program.

Separate caches may be maintained for instructions and for data operands. Information within a cache is maintained in contiguous bytes on an integral boundary called a cache block or cache line (or line, for short). A model may provide an EXTRACT CACHE ATTRIBUTE instruction which returns the size of a cache line in bytes. A model may also provide PREFETCH DATA and PREFETCH DATA RELATIVE LONG instructions which affects the prefetching of storage into the data or instruction cache or the releasing of data from the cache.

Storage is viewed as a long horizontal string of bits. For most operations, accesses to storage proceed in a left-to-right sequence. The string of bits is subdivided into units of eight bits. An eight-bit unit is called a byte, which is the basic building block of all information formats. Each byte location in storage is identified by a unique nonnegative integer, which is the address of that byte location or, simply, the byte address. Adjacent byte locations have consecutive addresses, starting with 0 on the left and proceeding in a left-to-right sequence. Addresses are unsigned binary integers and are 24, 31, or 64 bits.

Information is transmitted between storage and a CPU or a channel subsystem one byte, or a group of bytes, at a time. Unless otherwise specified, a group of bytes in storage is addressed by the leftmost byte of the group. The number of bytes in the group is either implied or explicitly specified by the operation to be performed. When used in a CPU operation, a group of bytes is called a field. Within each group of bytes, bits are numbered in a left-to-right sequence. The leftmost bits are sometimes referred to as the “high-order” bits and the rightmost bits as the “low-order” bits. Bit numbers are not storage addresses, however. Only bytes can be addressed. To operate on individual bits of a byte in storage, it is necessary to access the entire byte. The bits in a byte are numbered 0 through 7, from left to right. The bits in an address may be numbered 8-31 or 40-63 for 24-bit addresses or 1-31 or 33-63 for 31-bit addresses; they are numbered 0-63 for 64-bit addresses. Within any other fixed-length format of multiple bytes, the bits making up the format are consecutively numbered starting from 0. For purposes of error detection, and in preferably for correction, one or more check

bits may be transmitted with each byte or with a group of bytes. Such check bits are generated automatically by the machine and cannot be directly controlled by the program. Storage capacities are expressed in number of bytes. When the length of a storage-operand field is implied by the operation code of an instruction, the field is said to have a fixed length, which can be one, two, four, eight, or sixteen bytes. Larger fields may be implied for some instructions. When the length of a storage-operand field is not implied but is stated explicitly, the field is said to have a variable length. Variable-length operands can vary in length by increments of one byte. When information is placed in storage, the contents of only those byte locations are replaced that are included in the designated field, even though the width of the physical path to storage may be greater than the length of the field being stored.

Certain units of information must be on an integral boundary in storage. A boundary is called integral for a unit of information when its storage address is a multiple of the length of the unit in bytes. Special names are given to fields of 2, 4, 8, and 16 bytes on an integral boundary. A halfword is a group of two consecutive bytes on a two-byte boundary and is the basic building block of instructions. A word is a group of four consecutive bytes on a four-byte boundary. A doubleword is a group of eight consecutive bytes on an eight-byte boundary. A quadword is a group of 16 consecutive bytes on a 16-byte boundary. When storage addresses designate halfwords, words, doublewords, and quadwords, the binary representation of the address contains one, two, three, or four rightmost zero bits, respectively. Instructions must be on two-byte integral boundaries. The storage operands of most instructions do not have boundary-alignment requirements.

On models that implement separate caches for instructions and data operands, a significant delay may be experienced if the program stores into a cache line from which instructions are subsequently fetched, regardless of whether the store alters the instructions that are subsequently fetched.

INSTRUCTIONS:

Typically, operation of the CPU is controlled by instructions in storage that are executed sequentially, one at a time, left to right in an ascending sequence of storage addresses. A

change in the sequential operation may be caused by branching, LOAD PSW, interruptions, SIGNAL PROCESSOR orders, or manual intervention.

Preferably an instruction comprises two major parts:

- An operation code (op code), which specifies the operation to be performed
- Optionally, the designation of the operands that participate.

Instruction formats of the z/Architecture are shown in FIGs. 5A-5F. An instruction can simply provide an Opcode 501, or an opcode and a variety of fields including immediate operands or register specifiers for locating operands in registers or in memory. The Opcode can indicate to the hardware that implied resources (operands etc.) are to be used such as one or more specific general purpose registers (GPRs). Operands can be grouped in three classes: operands located in registers, immediate operands, and operands in storage. Operands may be either explicitly or implicitly designated. Register operands can be located in general, floating- point, access, or control registers, with the type of register identified by the op code. The register containing the operand is specified by identifying the register in a four-bit field, called the R field, in the instruction. For some instructions, an operand is located in an implicitly designated register, the register being implied by the op code. Immediate operands are contained within the instruction, and the 8-bit, 16-bit, or 32-bit field containing the immediate operand is called the I field. Operands in storage may have an implied length; be specified by a bit mask; be specified by a four-bit or eight-bit length specification, called the L field, in the instruction; or have a length specified by the contents of a general register.

The addresses of operands in storage are specified by means of a format that uses the contents of a general register as part of the address. This makes it possible to:

Specify a complete address by using an abbreviated notation

Perform address manipulation using instructions which employ general registers for operands

Modify addresses by program means without alteration of the instruction stream

Operate independent of the location of data areas by directly using addresses received from other programs.

The address used to refer to storage either is contained in a register designated by the R field in the instruction or is calculated from a base address, index, and displacement, specified by the B, X, and D fields, respectively, in the instruction. When the CPU is in the access-register mode, a B or R field may designate an access register in addition to being used to specify an address. To describe the execution of instructions, operands are preferably designated as first and second operands and, in some cases, third and fourth operands. In general, two operands participate in an instruction execution, and the result replaces the first operand.

An instruction is one, two, or three halfwords in length and must be located in storage on a halfword boundary. Referring to FIGs. 5A - 5F depicting instruction formats, each instruction is in one of 25 basic formats: E 501, I 502, RI 503 504, RIE 505 551 552 553 554, RIL 506 507, RIS 555, RR 510, RRE 511, RRF 512 513 514, RRS, RS 516 517, RSI 520, RSL 521, RSY 522 523, RX 524, RXE 525, RXF 526, RXY 527, S 530, SI 531, SIL 556, SIY 532, SS 533 534 535 536 537, SSE 541 and SSF 542, with three variations of RRF, two of RI, RIL, RS, and RSY, five of RIE and SS.

The format names indicate, in general terms, the classes of operands which participate in the operation and some details about fields:

- RIS denotes a register-and-immediate operation and a storage operation.
- RRS denotes a register-and-register operation and a storage operation.
- SIL denotes a storage-and-immediate operation, with a 16-bit immediate field.

In the I, RR, RS, RSI, RX, SI, and SS formats, the first byte of an instruction contains the op code. In the E, RRE, RRF, S, SIL, and SSE formats, the first two bytes of an instruction contain the op code, except that for some instructions in the S format, the op code is in only the first byte. In the RI and RIL formats, the op code is in the first byte and bit positions 12-15 of an instruction. In the RIE, RIS, RRS, RSL, RSY, RXE, RXF, RXY, and SIY formats, the op code is in the first byte and the sixth byte of an instruction. The first two bits of the first or only byte of the op code specify the length and format of the instruction, as follows:

In the RR, RRE, RRF, RRR, RX, RXE, RXF, RXY, RS, RSY, RSI, RI, RIE, and RIL formats, the contents of the register designated by the R1. field are called the first operand. The register containing the first operand is sometimes referred to as the “first operand location,” and sometimes as “register R1“. In the RR, RRE, RRF and RRR formats, the R2 field designates the register containing the second operand, and the R2 field may designate the same register as R1. In the RRF, RXF, RS, RSY, RSI, and RIE formats, the use of the R3 field depends on the instruction. In the RS and RSY formats, the R3 field may instead be an M3 field specifying a mask. The R field designates a general or access register in the general instructions, a general register in the control instructions, and a floating-point register or a general register in the floating-point instructions. For general and control registers, the register operand is in bit positions 32-63 of the 64-bit register or occupies the entire register, depending on the instruction.

In the I format, the contents of the eight-bit immediate- data field, the I field of the instruction, are directly used as the operand. In the SI format, the contents of the eight-bit immediate- data field, the I2 field of the instruction, are used directly as the second operand. The B1 and D1 fields specify the first operand, which is one byte in length. In the SIY format, the operation is the same except that DH1 and DL1 fields are used instead of a D1 field. In the RI format for the instructions ADD HALFWORD IMMEDIATE, COMPARE HALFWORD IMMEDIATE, LOAD HALFWORD IMMEDIATE, and MULTIPLY HALFWORD IMMEDIATE, the contents of the 16-bit I2 field of the instruction are used directly as a signed binary integer, and the R1 field specifies the first operand, which is 32 or 64 bits in length, depending on the instruction. For the instruction TEST UNDER MASK (TMHH, TMHL, TMLH, TMLL), the contents of the I2 field are used as a mask, and the R1 field specifies the first operand, which is 64 bits in length.

For the instructions INSERT IMMEDIATE, AND IMMEDIATE, OR IMMEDIATE, and LOAD LOGICAL IMMEDIATE, the contents of the I2 field are used as an unsigned binary integer or a logical value, and the R1 field specifies the first operand, which is 64 bits in length. For the relative-branch instructions in the RI and RSI formats, the contents of the 16-bit I2 field are used as a signed binary integer designating a number of halfwords. This number, when added to the address of the branch instruction, specifies the branch address.

For relative-branch instructions in the RIL format, the I2 field is 32 bits and is used in the same way.

For the relative-branch instructions in the RI and RSI formats, the contents of the 16-bit I2 field are used as a signed binary integer designating a number of halfwords. This number, when added to the address of the branch instruction, specifies the branch address. For relative-branch instructions in the RIL format, the I2 field is 32 bits and is used in the same way. For the RIE-format instructions COMPARE IMMEDIATE AND BRANCH RELATIVE and COMPARE LOGICAL IMMEDIATE AND BRANCH RELATIVE, the contents of the 8-bit I2 field is used directly as the second operand. For the RIE-format instructions COMPARE IMMEDIATE AND BRANCH, COMPARE IMMEDIATE AND TRAP, COMPARE LOGICAL IMMEDIATE AND BRANCH, and COMPARE LOGICAL IMMEDIATE AND TRAP, the contents of the 16-bit I2 field are used directly as the second operand. For the RIE-format instructions COMPARE AND BRANCH RELATIVE, COMPARE IMMEDIATE AND BRANCH RELATIVE, COMPARE LOGICAL AND BRANCH RELATIVE, and COMPARE LOGICAL IMMEDIATE AND BRANCH RELATIVE, the contents of the 16-bit I4 field are used as a signed binary integer designating a number of halfwords that are added to the address of the instruction to form the branch address.

For the RIL-format instructions ADD IMMEDIATE, ADD LOGICAL IMMEDIATE, ADD LOGICAL WITH SIGNED IMMEDIATE, COMPARE IMMEDIATE, COMPARE LOGICAL IMMEDIATE, LOAD IMMEDIATE, and MULTIPLY SINGLE IMMEDIATE, the contents of the 32-bit I2 field are used directly as a the second operand.

For the RIS-format instructions, the contents of the 8-bit I2 field are used directly as the second operand. In the SIL format, the contents of the 16-bit I2 field are used directly as the second operand. The B1 and D1 fields specify the first operand, as described below.

In the RSL, SI, SIL, SSE, and most SS formats, the contents of the general register designated by the B1 field are added to the contents of the D1 field to form the first-operand address. In the RS, RSY, S, SIY, SS, and SSE formats, the contents of the general register

designated by the B2 field are added to the contents of the D2 field or DH2 and DL2 fields to form the second-operand address. In the RX, RXE, RXF, and RXY formats, the contents of the general registers designated by the X2 and B2 fields are added to the contents of the D2 field or DH2 and DL2 fields to form the second-operand address. In the RIS and RRS formats, and in one SS format, the contents of the general register designated by the B4 field are added to the contents of the D4 field to form the fourth-operand address.

In the SS format with a single, eight-bit length field, for the instructions AND (NC), EXCLUSIVE OR (XC), MOVE (MVC), MOVE NUMERICS, MOVE ZONES, and OR (OC), L specifies the number of additional operand bytes to the right of the byte designated by the first-operand address. Therefore, the length in bytes of the first operand is 1-256, corresponding to a length code in L of 0-255. Storage results replace the first operand and are never stored outside the field specified by the address and length. In this format, the second operand has the same length as the first operand. There are variations of the preceding definition that apply to EDIT, EDIT AND MARK, PACK ASCII, PACK UNICODE, TRANSLATE, TRANSLATE AND TEST, UNPACK ASCII, and UNPACK UNICODE.

In the SS format with two length fields, and in the RSL format, L1 specifies the number of additional operand bytes to the right of the byte designated by the first-operand address. Therefore, the length in bytes of the first operand is 1-16, corresponding to a length code in L1 of 0-15. Similarly, L2 specifies the number of additional operand bytes to the right of the location designated by the second-operand address. Results replace the first operand and are never stored outside the field specified by the address and length. If the first operand is longer than the second, the second operand is extended on the left with zeros up to the length of the first operand. This extension does not modify the second operand in storage. In the SS format with two R fields, as used by the MOVE TO PRIMARY, MOVE TO SECONDARY, and MOVE WITH KEY instructions, the contents of the general register specified by the R1 field are a 32-bit unsigned value called the true length. The operands are both of a length called the effective length. The effective length is equal to the true length or 256, whichever is less. The instructions set the condition code to facilitate programming a loop to move the total number of bytes specified by the true length. The SS format with two R fields is also

used to specify a range of registers and two storage operands for the LOAD MULTIPLE DISJOINT instruction and to specify one or two registers and one or two storage operands for the PERFORM LOCKED OPERATION instruction.

A zero in any of the B1, B2, X2, or B4 fields indicates the absence of the corresponding address component. For the absent component, a zero is used informing the intermediate sum, regardless of the contents of general register 0. A displacement of zero has no special significance.

Bits 31 and 32 of the current PSW are the addressing- mode bits. Bit 31 is the extended-addressing mode bit, and bit 32 is the basic-addressing-mode bit. These bits control the size of the effective address produced by address generation. When bits 31 and 32 of the current PSW both are zeros, the CPU is in the 24-bit addressing mode, and 24-bit instruction and operand effective addresses are generated. When bit 31 of the current PSW is zero and bit 32 is one, the CPU is in the 31-bit addressing mode, and 31-bit instruction and operand effective addresses are generated. When bits 31 and 32 of the current PSW are both one, the CPU is in the 64-bit addressing mode, and 64-bit instruction and operand effective addresses are generated. Execution of instructions by the CPU involves generation of the addresses of instructions and operands.

When an instruction is fetched from the location designated by the current PSW, the instruction address is increased by the number of bytes in the instruction, and the instruction is executed. The same steps are then repeated by using the new value of the instruction address to fetch the next instruction in the sequence. In the 24-bit addressing mode, instruction addresses wrap around, with the halfword at instruction address $2^{24} - 2$ being followed by the halfword at instruction address 0. Thus, in the 24-bit addressing mode, any carry out of PSW bit position 104, as a result of updating the instruction address, is lost. In the 31-bit or 64-bit addressing mode, instruction addresses similarly wrap around, with the halfword at instruction address $2^{31} - 2$ or $2^{64} - 2$, respectively, followed by the halfword at instruction address 0. A carry out of PSW bit position 97 or 64, respectively, is lost.

An operand address that refers to storage is derived from an intermediate value, which either is contained in a register designated by an R field in the instruction or is calculated from the sum of three binary numbers: base address, index, and displacement. The base address (B) is a 64-bit number contained in a general register specified by the program in a four bit field, called the B field, in the instruction. Base addresses can be used as a means of independently addressing each program and data area. In array type calculations, it can designate the location of an array, and, in record-type processing, it can identify the record. The base address provides for addressing the entire storage. The base address may also be used for indexing.

The index (X) is a 64-bit number contained in a general register designated by the program in a four-bit field, called the X field, in the instruction. It is included only in the address specified by the RX-, RXE-, and RXY-format instructions. The RX-, RXE-, RXY-, and RXY-format instructions permit double indexing; that is, the index can be used to provide the address of an element within an array.

The displacement (D) is a 12-bit or 20-bit number contained in a field, called the D field, in the instruction. A 12-bit displacement is unsigned and provides for relative addressing of up to 4,095 bytes beyond the location designated by the base address. A 20-bit displacement is signed and provides for relative addressing of up to 524,287 bytes beyond the base address location or of up to 524,288 bytes before it. In array-type calculations, the displacement can be used to specify one of many items associated with an element. In the processing of records, the displacement can be used to identify items within a record. A 12-bit displacement is in bit positions 20-31 of instructions of certain formats. In instructions of some formats, a second 12-bit displacement also is in the instruction, in bit positions 36-47.

A 20-bit displacement is in instructions of only the RSY, RXY, or SIY format. In these instructions, the D field consists of a DL (low) field in bit positions 20-31 and of a DH (high) field in bit positions 32-39. When the long-displacement facility is installed, the numeric value of the displacement is formed by appending the contents of the DH field on the left of the contents of the DL field. When the long-displacement facility is not installed,

the numeric value of the displacement is formed by appending eight zero bits on the left of the contents of the DL field, and the contents of the DH field are ignored.

In forming the intermediate sum, the base address and index are treated as 64-bit binary integers. A 12-bit displacement is treated as a 12-bit unsigned binary integer, and 52 zero bits are appended on the left. A 20-bit displacement is treated as a 20-bit signed binary integer, and 44 bits equal to the sign bit are appended on the left. The three are added as 64-bit binary numbers, ignoring overflow. The sum is always 64 bits long and is used as an intermediate value to form the generated address. The bits of the intermediate value are numbered 0-63. A zero in any of the B1, B2, X2, or B4 fields indicates the absence of the corresponding address component. For the absent component, a zero is used in forming the intermediate sum, regardless of the contents of general register 0. A displacement of zero has no special significance.

When an instruction description specifies that the contents of a general register designated by an R field are used to address an operand in storage, the register contents are used as the 64-bit intermediate value.

An instruction can designate the same general register both for address computation and as the location of an operand. Address computation is completed before registers, if any, are changed by the operation. Unless otherwise indicated in an individual instruction definition, the generated operand address designates the leftmost byte of an operand in storage.

The generated operand address is always 64 bits long, and the bits are numbered 0-63. The manner in which the generated address is obtained from the intermediate value depends on the current addressing mode. In the 24-bit addressing mode, bits 0-39 of the intermediate value are ignored, bits 0-39 of the generated address are forced to be zeros, and bits 40-63 of the intermediate value become bits 40-63 of the generated address. In the 31-bit addressing mode, bits 0-32 of the intermediate value are ignored, bits 0-32 of the generated address are forced to be zero, and bits 33-63 of the intermediate value become bits 33-63 of the generated address. In the 64-bit addressing mode, bits 0-63 of the intermediate value become bits 0-63 of the generated address. Negative values may be used in index and base-address

registers. Bits 0-32 of these values are ignored in the 31-bit addressing mode, and bits 0-39 are ignored in the 24-bit addressing mode.

For branch instructions, the address of the next instruction to be executed when the branch is taken is called the branch address. Depending on the branch instruction, the instruction format may be RR, RRE, RX, RXY, RS, RSY, RSI, RI, RIE, or RIL. In the RS, RSY, RX, and RXY formats, the branch address is specified by a base address, a displacement, and, in the RX and RXY formats, an index. In these formats, the generation of the intermediate value follows the same rules as for the generation of the operand-address intermediate value. In the RR and RRE formats, the contents of the general register designated by the R2 field are used as the intermediate value from which the branch address is formed. General register 0 cannot be designated as containing a branch address. A value of zero in the R2 field causes the instruction to be executed without branching.

The relative-branch instructions are in the RSI, RI, RIE, and RIL formats. In the RSI, RI, and RIE formats for the relative-branch instructions, the contents of the I2 field are treated as a 16-bit signed binary integer designating a number of halfwords. In the RIL format, the contents of the I2 field are treated as a 32-bit signed binary integer designating a number of halfwords. The branch address is the number of halfwords designated by the I2 field added to the address of the relative-branch instruction.

The 64-bit intermediate value for a relative branch instruction in the RSI, RI, RIE, or RIL format is the sum of two addends, with overflow from bit position 0 ignored. In the RSI, RI, or RIE format, the first addend is the contents of the I2 field with one zero bit appended on the right and 47 bits equal to the sign bit of the contents appended on the left, except that for COMPARE AND BRANCH RELATIVE, COMPARE IMMEDIATE AND BRANCH RELATIVE, COMPARE LOGICAL AND BRANCH RELATIVE and COMPARE LOGICAL IMMEDIATE AND BRANCH RELATIVE, the first addend is the contents of the I4 field, with bits appended as described above for the I2 field. In the RIL format, the first addend is the contents of the I2 field with one zero bit appended on the right and 31 bits equal to the sign bit of the contents appended on the left. In all formats, the second addend is the 64-bit address of the branch instruction. The address of the branch instruction is the

instruction address in the PSW before that address is updated to address the next sequential instruction, or it is the address of the target of the EXECUTE instruction if EXECUTE is used. If EXECUTE is used in the 24-bit or 31-bit addressing mode, the address of the branch instruction is the target address with 40 or 33 zeros, respectively, appended on the left.

The branch address is always 64 bits long, with the bits numbered 0-63. The branch address replaces bits 64-127 of the current PSW. The manner in which the branch address is obtained from the intermediate value depends on the addressing mode. For those branch instructions which change the addressing mode, the new addressing mode is used. In the 24-bit addressing mode, bits 0-39 of the intermediate value are ignored, bits 0-39 of the branch address are made zeros, and bits 40-63 of the intermediate value become bits 40-63 of the branch address. In the 31-bit addressing mode, bits 0-32 of the intermediate value are ignored, bits 0-32 of the branch address are made zeros, and bits 33-63 of the intermediate value become bits 33-63 of the branch address. In the 64-bit addressing mode, bits 0-63 of the intermediate value become bits 0-63 of the branch address.

For several branch instructions, branching depends on satisfying a specified condition. When the condition is not satisfied, the branch is not taken, normal sequential instruction execution continues, and the branch address is not used. When a branch is taken, bits 0-63 of the branch address replace bits 64-127 of the current PSW. The branch address is not used to access storage as part of the branch operation. A specification exception due to an odd branch address and access exceptions due to fetching of the instruction at the branch location are not recognized as part of the branch operation but instead are recognized as exceptions associated with the execution of the instruction at the branch location.

A branch instruction, such as BRANCH AND SAVE, can designate the same general register for branch address computation and as the location of an operand. Branch-address computation is completed before the remainder of the operation is performed.

The program-status word (PSW), described in Chapter 4 “Control” contains information required for proper program execution. The PSW is used to control instruction sequencing and to hold and indicate the status of the CPU in relation to the program currently being

executed. The active or controlling PSW is called the current PSW. Branch instructions perform the functions of decision making, loop control, and subroutine linkage. A branch instruction affects instruction sequencing by introducing a new instruction address into the current PSW. The relative-branch instructions with a 16-bit I2 field allow branching to a location at an offset of up to plus 64K - 2 bytes or minus 64K bytes relative to the location of the branch instruction, without the use of a base register. The relative-branch instructions with a 32-bit I2 field allow branching to a location at an offset of up to plus 4G - 2 bytes or minus 4G bytes relative to the location of the branch instruction, without the use of a base register.

Facilities for decision making are provided by the **BRANCH ON CONDITION**, **BRANCH RELATIVE ON CONDITION**, and **BRANCH RELATIVE ON CONDITION LONG** instructions. These instructions inspect a condition code that reflects the result of a majority of the arithmetic, logical, and I/O operations. The condition code, which consists of two bits, provides for four possible condition-code settings: 0, 1, 2, and 3.

The specific meaning of any setting depends on the operation that sets the condition code. For example, the condition code reflects such conditions as zero, nonzero, first operand high, equal, overflow, and subchannel busy. Once set, the condition code remains unchanged until modified by an instruction that causes a different condition code to be set.

Loop control can be performed by the use of **BRANCH ON CONDITION**, **BRANCH RELATIVE ON CONDITION**, and **BRANCH RELATIVE ON CONDITION LONG** to test the outcome of address arithmetic and counting operations. For some particularly frequent combinations of arithmetic and tests, **BRANCH ON COUNT**, **BRANCH ON INDEX HIGH**, and **BRANCH ON INDEX LOW OR EQUAL** are provided, and relative-branch equivalents of these instructions are also provided. These branches, being specialized, provide increased performance for these tasks.

Subroutine linkage when a change of the addressing mode is not required is provided by the **BRANCH AND LINK** and **BRANCH AND SAVE** instructions. (This discussion of **BRANCH AND SAVE** applies also to **BRANCH RELATIVE AND SAVE** and **BRANCH**

RELATIVE AND SAVE LONG.) Both of these instructions permit not only the introduction of a new instruction address but also the preservation of a return address and associated information. The return address is the address of the instruction following the branch instruction in storage, except that it is the address of the instruction following an EXECUTE instruction that has the branch instruction as its target.

Both BRANCH AND LINK and BRANCH AND SAVE have an R1 field. They form a branch address by means of fields that depend on the instruction. The operations of the instructions are summarized as follows:

- In the 24-bit addressing mode, both instructions place the return address in bit positions 40-63 of general register R1 and leave bits 0-31 of that register unchanged. BRANCH AND LINK places the instruction-length code for the instruction and also the condition code and program mask from the current PSW in bit positions 32-39 of general register R1. BRANCH AND SAVE places zeros in those bit positions.
- In the 31-bit addressing mode, both instructions place the return address in bit positions 33-63 and a one in bit position 32 of general register R1, and they leave bits 0-31 of the register unchanged.
- In the 64-bit addressing mode, both instructions place the return address in bit positions 0-63 of general register R1.
- In any addressing mode, both instructions generate the branch address under the control of the current addressing mode. The instructions place bits 0-63 of the branch address in bit positions 64-127 of the PSW. In the RR format, both instructions do not perform branching if the R2 field of the instruction is zero.

It can be seen that, in the 24-bit or 31-bit addressing mode, BRANCH AND SAVE places the basic addressing- mode bit, bit 32 of the PSW, in bit position 32 of general register R1. BRANCH AND LINK does so in the 31-bit addressing mode. The instructions BRANCH AND SAVE AND SET MODE and BRANCH AND SET MODE are for use when a change of the addressing mode is required during linkage. These instructions have R1 and R2 fields. The operations of the instructions are summarized as follows:

- **BRANCH AND SAVE AND SET MODE** sets the contents of general register R1 the same as **BRANCH AND SAVE**. In addition, the instruction places the extended-addressing-mode bit, bit 31 of the PSW, in bit position 63 of the register.
- **BRANCH AND SET MODE**, if R1 is nonzero, performs as follows. In the 24- or 31-bit mode, it places bit 32 of the PSW in bit position 32 of general register R1, and it leaves bits 0-31 and 33-63 of the register unchanged. Note that bit 63 of the register should be zero if the register contains an instruction address. In the 64-bit mode, the instruction places bit 31 of the PSW (a one) in bit position 63 of general register R1, and it leaves bits 0-62 of the register unchanged.
- When R2 is nonzero, both instructions set the addressing mode and perform branching as follows. Bit 63 of general register R2 is placed in bit position 31 of the PSW. If bit 63 is zero, bit 32 of the register is placed in bit position 32 of the PSW. If bit 63 is one, PSW bit 32 is set to one. Then the branch address is generated from the contents of the register, except with bit 63 of the register treated as a zero, under the control of the new addressing mode. The instructions place bits 0-63 of the branch address in bit positions 64-127 of the PSW. Bit 63 of general register R2 remains unchanged and, therefore, may be one upon entry to the called program. If R2 is the same as R1, the results in the designated general register are as specified for the R1 register.

INTERRUPTIONS (CONTEXT SWITCH):

The interruption mechanism permits the CPU to change its state as a result of conditions external to the configuration, within the configuration, or within the CPU itself. To permit fast response to conditions of high priority and immediate recognition of the type of condition, interruption conditions are grouped into six classes: external, input/output, machine check, program, restart, and supervisor call.

An interruption consists in storing the current PSW as an old PSW, storing information identifying the cause of the interruption, and fetching a new PSW. Processing resumes as specified by the new PSW. The old PSW stored on an interruption normally contains the address of the instruction that would have been executed next had the interruption not occurred, thus permitting resumption of the interrupted program. For program and supervisor-call interruptions, the information stored also contains a code that identifies the

length of the last-executed instruction, thus permitting the program to respond to the cause of the interruption. In the case of some program conditions for which the normal response is re-execution of the instruction causing the interruption, the instruction address directly identifies the instruction last executed.

Except for restart, an interruption can occur only when the CPU is in the operating state. The restart interruption can occur with the CPU in either the stopped or operating state.

Any access exception is generated as part of the execution of the instruction with which the exception is associated. An access exception is not generated when the CPU attempts to prefetch from an unavailable location or detects some other access-exception condition, but a branch instruction or an interruption changes the instruction sequence such that the instruction is not executed. Every instruction can cause an access exception to be generated because of instruction fetch. Additionally, access exceptions associated with instruction execution may occur because of an access to an operand in storage. An access exception due to fetching an instruction is indicated when the first instruction halfword cannot be fetched without encountering the exception. When the first halfword of the instruction has no access exceptions, access exceptions may be indicated for additional halfwords according to the instruction length specified by the first two bits of the instruction; however, when the operation can be performed without accessing the second or third halfwords of the instruction, it is unpredictable whether the access exception is indicated for the unused part. Since the indication of access exceptions for instruction fetch is common to all instructions, it is not covered in the individual instruction definitions.

Except where otherwise indicated in the individual instruction description, the following rules apply for exceptions associated with an access to an operand location. For a fetch-type operand, access exceptions are necessarily indicated only for that portion of the operand which is required for completing the operation. It is unpredictable whether access exceptions are indicated for those portions of a fetch-type operand which are not required for completing the operation.

For a store-type operand, access exceptions are generated for the entire operand even if the operation could be completed without the use of the inaccessible part of the operand. In situations where the value of a store-type operand is defined to be unpredictable, it is unpredictable whether an access exception is indicated. Whenever an access to an operand location can cause an access exception to be generated, the word “access” is included in the list of program exceptions in the description of the instruction. This entry also indicates which operand can cause the exception to be generated and whether the exception is generated on a fetch or store access to that operand location. Access exceptions are generated only for the portion of the operand as defined for each particular instruction.

An operation exception is generated when the CPU attempts to execute an instruction with an invalid operation code. The operation code may be unassigned, or the instruction with that operation code may not be installed on the CPU. The operation is suppressed. The instruction-length code is 1, 2, or 3. The operation exception is indicated by a program interruption code of 0001 hex (or 0081 hex if a concurrent PER event is indicated).

Some models may offer instructions not described in this publication, such as those provided for assists or as part of special or custom features. Consequently, operation codes not described in this publication do not necessarily cause an operation exception to be generated. Furthermore, these instructions may cause modes of operation to be set up or may otherwise alter the machine so as to affect the execution of subsequent instructions. To avoid causing such an operation, an instruction with an operation code not described in this publication should be executed only when the specific function associated with the operation code is desired.

A specification exception is generated when any of the following is true:

1. A one is introduced into an unassigned bit position of the PSW (that is, any of bit positions 0, 2-4, 24-30, or 33-63). This is handled as an early PSW specification exception.
2. A one is introduced into bit position 12 of the PSW. This is handled as an early PSW specification exception.
3. The PSW is invalid in any of the following ways: a. Bit 31 of the PSW is one and bit 32 is zero. b. Bits 31 and 32 of the PSW are zero, indicating the 24-bit addressing mode, and bits

64-103 of the PSW are not all zeros. c. Bit 31 of the PSW is zero and bit 32 is one, indicating the 31-bit addressing mode, and bits 64-96 of the PSW are not all zeros. This is handled as an early PSW specification exception.

4. The PSW contains an odd instruction address.

5. An operand address does not designate an integral boundary in an instruction requiring such integral-boundary designation.

6. An odd-numbered general register is designated by an R field of an instruction that requires an even-numbered register designation.

7. A floating-point register other than 0, 1, 4, 5, 8, 9, 12, or 13 is designated for an extended operand.

8. The multiplier or divisor in decimal arithmetic exceeds 15 digits and sign.

9. The length of the first-operand field is less than or equal to the length of the second-operand field in decimal multiplication or division.

10. Execution of CIPHER MESSAGE, CIPHER MESSAGE WITH CHAINING, COMPUTE INTERMEDIATE MESSAGE DIGEST, COMPUTE LAST MESSAGE DIGEST, or COMPUTE MESSAGE AUTHENTICATION CODE is attempted, and the function code in bits 57-63 of general register 0 contain an unassigned or uninstalled function code.

11. Execution of CIPHER MESSAGE or CIPHER MESSAGE WITH CHAINING is attempted, and the R1 or R2 field designates an odd-numbered register or general register 0.

12. Execution of CIPHER MESSAGE, CIPHER MESSAGE WITH CHAINING, COMPUTE INTERMEDIATE MESSAGE DIGEST or COMPUTE MESSAGE AUTHENTICATION CODE is attempted, and the second operand length is not a multiple of the data block size of the designated function. This specification-exception condition does not apply to the query functions.

13. Execution of COMPARE AND FORM CODEWORD is attempted, and general registers 1, 2, and 3 do not initially contain even values.

32. Execution of COMPARE AND SWAP AND STORE is attempted and any of the following conditions exist:

- The function code specifies an unassigned value.
- The store characteristic specifies an unassigned value.
- The function code is 0, and the first operand is not designated on a word boundary.

- The function code is 1, and the first operand is not designated on a doubleword boundary.
- The second operand is not designated on an integral boundary corresponding to the size of the store value.

33. Execution of COMPARE LOGICAL LONG UNICODE or MOVE LONG UNICODE is attempted, and the contents of either general register R1 + 1 or R3 + 1 do not specify an even number of bytes.

34. Execution of COMPARE LOGICAL STRING, MOVE STRING or SEARCH STRING is attempted, and bits 32-55 of general register 0 are not all zeros.

35. Execution of COMPRESSION CALL is attempted, and bits 48-51 of general register 0 have any of the values 0000 and 0110-1111 binary.

36. Execution of COMPUTE INTERMEDIATE MESSAGE DIGEST, COMPUTE LAST MESSAGE DIGEST, or COMPUTE MESSAGE AUTHENTICATION CODE is attempted, and either of the following is true:

- The R2 field designates an odd-numbered register or general register 0.
- Bit 56 of general register 0 is not zero.

37. Execution of CONVERT HFP TO BFP, CONVERT TO FIXED (BFP or HFP), or LOAD FP INTEGER (BFP) is attempted, and the M3 field does not designate a valid modifier.

38. Execution of DIVIDE TO INTEGER is attempted, and the M4 field does not designate a valid modifier.

39. Execution of EXECUTE is attempted, and the target address is odd.

40. Execution of EXTRACT STACKED STATE is attempted, and the code in bit positions 56-63 of general register R2 is greater than 4 when the ASN-and-LX-reuse facility is not installed or is greater than 5 when the facility is installed.

41. Execution of FIND LEFTMOST ONE is attempted, and the R1 field designates an oddnumbered register.

42. Execution of INVALIDATE DAT TABLE ENTRY is attempted, and bits 44-51 of general register R2 are not all zeros.

43. Execution of LOAD FPC is attempted, and one or more bits of the second operand corresponding to unsupported bits in the FPC register are one.

44. Execution of LOAD PAGE-TABLE-ENTRY ADDRESS is attempted and the M4 field of the instruction contains any value other than 0000-0100 binary.

45. Execution of LOAD PSW is attempted and bit 12 of the doubleword at the second-operand address is zero. It is model dependent whether or not this exception is generated.
46. Execution of MONITOR CALL is attempted, and bit positions 8-11 of the instruction do not contain zeros.
47. Execution of MOVE PAGE is attempted, and bit positions 48-51 of general register 0 do not contain zeros or bits 52 and 53 of the register are both one.
48. Execution of PACK ASCII is attempted, and the L2 field is greater than 31.
49. Execution of PACK UNICODE is attempted, and the L2 field is greater than 63 or is even.
50. Execution of PERFORM FLOATING POINT OPERATION is attempted, bit 32 of general register 0 is zero, and one or more fields in bits 33- 63 are invalid or designate an uninstalled function.
51. Execution of PERFORM LOCKED OPERATION is attempted, and any of the following is true: • The T bit, bit 55 of general register 0 is zero, and the function code in bits 56-63 of the register is invalid. • Bits 32-54 of general register 0 are not all zeros. • In the access-register mode, for function codes that cause use of a parameter list containing an ALET, the R3 field is zero.
52. Execution of PERFORM TIMING FACILITY FUNCTION is attempted, and either of the following is true: • Bit 56 of general register 0 is not zero. • Bits 57-63 of general register 0 specify an unassigned or uninstalled function code.
53. Execution of PROGRAM TRANSFER or PROGRAM TRANSFER WITH INSTANCE is attempted, and all of the following are true: • The extended-addressing-mode bit in the PSW is zero. • The basic-addressing-mode bit, bit 32, in the general register designated by the R2 field of the instruction is zero. • Bits 33-39 of the instruction address in the same register are not all zeros.
54. Execution of RESUME PROGRAM is attempted, and either of the following is true:
- Bits 31, 32, and 64-127 of the PSW field in the second operand are not valid for placement in the current PSW. The exception is generated if any of the following is true: – Bits 31 and 32 are both zero and bits 64-103 are not all zeros. – Bits 31 and 32 are zero and one, respectively, and bits 64-96 are not all zeros. – Bits 31 and 32 are one and zero, respectively. – Bit 127 is one.
 - Bits 0-12 of the parameter list are not all zeros.

55. Execution of SEARCH STRING UNICODE is attempted, and bits 32-47 of general register 0 are not all zeros.
56. Execution of SET ADDRESS SPACE CONTROL or SET ADDRESS SPACE CONTROL FAST is attempted, and bits 52 and 53 of the second-operand address are not both zeros.
57. Execution of SET ADDRESSING MODE (SAM24) is attempted, and bits 0-39 of the un-updated instruction address in the PSW, bits 64-103 of the PSW, are not all zeros.
58. Execution of SET ADDRESSING MODE (SAM31) is attempted, and bits 0-32 of the un-updated instruction address in the PSW, bits 64-96 of the PSW, are not all zeros.
59. Execution of SET CLOCK PROGRAMMABLE FIELD is attempted, and bits 32-47 of general register 0 are not all zeros.
60. Execution of SET FPC is attempted, and one or more bits of the first operand corresponding to unsupported bits in the FPC register are one.
61. Execution of STORE SYSTEM INFORMATION is attempted, the function code in general register 0 is valid, and either of the following is true: • Bits 36-55 of general register 0 and bits 32- 47 of general register 1 are not all zeros. • The second-operand address is not aligned on a 4K-byte boundary.
62. Execution of TRANSLATE TWO TO ONE or TRANSLATE TWO TO TWO is attempted, and the length in general register R1 + 1 does not specify an even number of bytes.
63. Execution of UNPACK ASCII is attempted, and the L1 field is greater than 31.
64. Execution of UNPACK UNICODE is attempted, and the L1 field is greater than 63 or is even.
65. Execution of UPDATE TREE is attempted, and the initial contents of general registers 4 and 5 are not a multiple of 8 in the 24-bit or 31-bit addressing mode or are not a multiple of 16 in the 64-bit addressing mode. The execution of the instruction identified by the old PSW is suppressed. However, for early PSW specification exceptions (causes 1-3) the operation that introduces the new PSW is completed, but an interruption occurs immediately thereafter. Preferably, the instruction-length code (ILC) is 1, 2, or 3, indicating the length of the instruction causing the exception. When the instruction address is odd (cause 4 on page 6-33), it is unpredictable whether the ILC is 1, 2, or 3. When the exception is generated because of an early PSW specification exception (causes 1-3) and the exception has been

introduced by LOAD PSW, LOAD PSW EXTENDED, PROGRAM RETURN, or an interruption, the ILC is 0. When the exception is introduced by SET ADDRESSING MODE (SAM24, SAM31), the ILC is 1, or it is 2 if SET ADDRESSING MODE was the target of EXECUTE. When the exception is introduced by SET SYSTEM MASK or by STORE THEN OR SYSTEM MASK, the ILC is 2.

Program interruptions are used to report exceptions and events which occur during execution of the program. A program interruption causes the old PSW to be stored at real locations 336-351 and a new PSW to be fetched from real locations 464-479. The cause of the interruption is identified by the interruption code. The interruption code is placed at real locations 142-143, the instruction-length code is placed in bit positions 5 and 6 of the byte at real location 141 with the rest of the bits set to zeros, and zeros are stored at real location 140. For some causes, additional information identifying the reason for the interruption is stored at real locations 144-183. If the PER-3 facility is installed, then, as part of the program interruption action, the contents of the breaking-event-address register are placed in real storage locations 272-279. Except for PER events and the crypto-operation exception, the condition causing the interruption is indicated by a coded value placed in the rightmost seven bit positions of the interruption code. Only one condition at a time can be indicated. Bits 0-7 of the interruption code are set to zeros. PER events are indicated by setting bit 8 of the interruption code to one. When this is the only condition, bits 0-7 and 9-15 are also set to zeros. When a PER event is indicated concurrently with another program interruption condition, bit 8 is one, and bits 0-7 and 9-15 are set as for the other condition. The crypto-operation exception is indicated by an interruption code of 0119 hex, or 0199 hex if a PER event is also indicated.

When there is a corresponding mask bit, a program interruption can occur only when that mask bit is one. The program mask in the PSW controls four of the exceptions, the IEEE masks in the FPC register control the IEEE exceptions, bit 33 in control register 0 controls whether SET SYSTEM MASK causes a special- operation exception, bits 48-63 in control register 8 control interruptions due to monitor events, and a hierarchy of masks control interruptions due to PER events. When any controlling mask bit is zero, the condition is ignored; the condition does not remain pending.

When the new PSW for a program interruption has a PSW-format error or causes an exception to be generated in the process of instruction fetching, a string of program interruptions may occur.

Some of the conditions indicated as program exceptions may be generated also by the channel subsystem, in which case the exception is indicated in the subchannel-status word or extended-status word.

When a data exception causes a program interruption, a data-exception code (DXC) is stored at location 147, and zeros are stored at locations 144-146. The DXC distinguishes between the various types of data-exception conditions. When the AFP-register (additional floating-point register) control bit, bit 45 of control register 0, is one, the DXC is also placed in the DXC field of the floating-point-control (FPC) register. The DXC field in the FPC register remains unchanged when any other program exception is reported. The DXC is an 8-bit code indicating the specific cause of a data exception.

DXC 2 and 3 are mutually exclusive and are of higher priority than any other DXC. Thus, for example, DXC 2 (BFP instruction) takes precedence over any IEEE exception; and DXC 3 (DFP instruction) takes precedence over any IEEE exception or simulated IEEE exception. As another example, if the conditions for both DXC 3 (DFP instruction) and DXC 1 (AFP register) exist, DXC 3 is reported. When both a specification exception and an AFP register data exception apply, it is unpredictable which one is reported.

An addressing exception is generated when the CPU attempts to reference a main-storage location that is not available in the configuration. A main-storage location is not available in the configuration when the location is not installed, when the storage unit is not in the configuration, or when power is off in the storage unit. An address designating a storage location that is not available in the configuration is referred to as invalid. The operation is suppressed when the address of the instruction is invalid. Similarly, the operation is suppressed when the address of the target instruction of EXECUTE is invalid. Also, the unit of operation is suppressed when an addressing exception is encountered in accessing a table or table entry. The tables and table entries to which the rule applies are the dispatchable-unit-

control table, the primary ASN second- table entry, and entries in the access list, region first table, region second table, region third table, segment table, page table, linkage table, linkage- first table, linkage-second table, entry table, ASN first table, ASN second table, authority table, linkage stack, and trace table. Addressing exceptions result in suppression when they are encountered for references to the region first table, region second table, region third table, segment table, and page table, in both implicit references for dynamic address translation and references associated with the execution of LOAD PAGE-TABLE-ENTRY ADDRESS, LOAD REAL ADDRESS, STORE REAL ADDRESS, and TEST PROTECTION. Similarly, addressing exceptions for accesses to the dispatchable-unit control table, primary ASN-second-table entry, access list, ASN second table, or authority table result in suppression when they are encountered in access-register translation done either implicitly or as part of LOAD PAGE-TABLE-ENTRY ADDRESS, LOAD REAL ADDRESS, STORE REAL ADDRESS, TEST ACCESS, or TEST PROTECTION. Except for some specific instructions whose execution is suppressed, the operation is terminated for an operand address that can be translated but designates an unavailable location. For termination, changes may occur only to result fields. In this context, the term “result field” includes the condition code, registers, and any storage locations that are provided and that are designated to be changed by the instruction.

The foregoing is useful in understanding the terminology and structure of one computer system embodiment. Embodiments not limited to the z/Architecture or to the description provided thereof. Embodiments can be advantageously applied to other computer architectures of other computer manufacturers with the teaching herein.

Referring to FIG. 7, a computer system may be running an Operating System (OS) 701 and two or more application programs 702 703. Context switching is employed to permit an OS to manage resources used by applications. In one example, an OS 701 sets an interrupt timer and initiates 704 a context switch action in order to permit an application program to run for a period specified by the interrupt timer. The context switch action saves 705 State Information of the OS including the program counter of the OS pointing to a next OS instruction to be executed. The context switch action next obtains 705 State Information of Application Program #1 702 to permit 706 the application program #1 702 to start executing

instructions at the Application Programs obtained current program counter. When the interrupt timer expires, a context switch 704 action is initiated to return the computer system to the OS.

Different processor architectures provide a limited number of general registers (GRs), sometimes referred to as general purpose registers, that are explicitly (and/or implicitly) identified by instructions of the architected instruction set. IBM z/Architecture and its predecessor architectures (dating back to the original System 360 circa 1964) provide 16 general registers (GRs) for each central processing unit (CPU). GRs may be used by processors (central processing unit (CPU)) instructions as follows:

As a source operand of an arithmetic or logical operation.

As a target operand of an arithmetic or logical operation.

As a the address of a memory operand (either a base register, index register, or directly).

As the length of a memory operand.

Other uses such as providing a function code or other information to and from an instruction.

Until the introduction of the IBM z/Architecture mainframe in 2000, a mainframe general register consisted of 32 bits; with the introduction of z/Architecture, a general register consisted of 64 bits, however, for compatibility reasons, many z/Architecture instructions continue to support 32 bits.

Similarly, other architectures, such as the x86 from Intel® for example, provide compatibility modes such that a current machine, having, for example 32 bit registers, provide modes for instructions to access only the first 8 bits or 16 bits of the 32 bit GR.

Even in early IBM System 360 environments, 16 registers (identified by a 4 bit register field in an instruction for example) proved to be daunting to assembler programmers and compiler designers. A moderately-size program could require several base registers to address code and data, limiting the number of registers available to hold active variables. Certain techniques have been used to address the limited number of registers:

Program design (as simple as modular programming) helped to minimize base-register overutilization.

Compilers have used techniques such as register "coloring" to manage the dynamic reassignment of registers.

Base register usage can be reduced with the following:

Newer arithmetic and logical instructions with immediate constants (within the instruction).

Newer instructions with relative-immediate operand addresses.

Newer instructions with long displacements.

However, there remains constant register pressure when there are more live variables and addressing scope than can be accommodated by the number of registers in the CPU.

z/Architecture provides three program-selectable addressing modes: 24-, 31-, and 64-bit addressing. However, for programs that neither require 64-bit values nor exploit 64-bit memory addressing, having 64-bit GRs is of limited benefit. The following disclosure describes a technique of exploiting 64-bit registers for programs that do not generally use 64-bit addressing or variables.

Within this disclosure, a convention is used where bit positions of registers are numbered in ascending order from left to right (Big Endian). In a 64-bit register, bit 0 (the leftmost bit) represents the most significant value (2^{63}) and bit 63 (the rightmost bit) represents the least significant value (2^0). The leftmost 32 bits of such a register (bits 0-31) are called the high word, and the rightmost 32 bits of the register (bits 32-63) are called the low word where a word is 32 bits.

INTERLOCKED-ACCESS FACILITY:

In an example z/Architecture embodiment, an interlocked-access facility may be available that provides the means by which a load, update, and store operation can be performed with interlocked update in a single instruction (as opposed to using a compare-and-swap type of

update). The facility also provides an instruction to attempt to load from two distinct storage locations in an interlocked-fetch manner. The facility provides the following instructions

- LOAD AND ADD
- LOAD AND ADD LOGICAL
- LOAD AND AND
- LOAD AND EXCLUSIVE OR
- LOAD AND OR
- LOAD PAIR DISJOINT

LOAD/STORE ON CONDITION FACILITY:

In an example z/Architecture embodiment, a load/store-on-condition facility may provide the means by which selected operations may be executed only when a condition-code-mask field of the instruction matches the current condition code in the PSW. The facility provides the following instructions.

- LOAD ON CONDITION
- STORE ON CONDITION

DISTINCT-OPERANDS FACILITY:

In an example z/Architecture embodiment, a distinct-operands facility may be provide alternate forms of selected arithmetic and logical instructions in which the result register may be different from either of the source registers. The facility provides alternate forms for the following instructions.

- ADD
- ADD IMMEDIATE
- ADD LOGICAL
- ADD LOGICAL WITH SIGNED IMMEDIATE
- AND
- EXCLUSIVE OR
- OR
- SHIFT LEFT SINGLE
- SHIFT LEFT SINGLE LOGICAL
- SHIFT RIGHT SINGLE

- SHIFT RIGHT SINGLE LOGICAL
- SUBTRACT
- SUBTRACT LOGICAL

POPULATION-COUNT FACILITY

In an example z/Architecture embodiment, a population-count facility may provide the POPULATION COUNT instruction which provides a count of one bits in each byte of a general register.

STORAGE-OPERAND REFERENCES:

For certain special instructions, the fetch references for multiple operands may appear to be interlocked against certain accesses by other CPUs and by channel programs. Such an fetch reference is called an interlocked-fetch reference. The fetch accesses associated with an interlocked-fetch reference do not necessarily occur one immediately after the other, but store accesses by other CPUs may not occur at the same locations as the interlocked-fetch reference between the fetch accesses of the interlocked fetch reference. The storage-operand fetch reference for the LOADPAIR DISJOINT instruction may be an interlocked-fetch reference. Whether or not LOADPAIR DISJOINT is able to fetch both operands by means of an interlocked fetch is indicated by the condition code. For certain special instructions, the update reference is interlocked against certain accesses by other CPUs and channel programs. Such an update reference is called an interlocked-update reference. The fetch and store accesses associated with an interlocked-update reference do not necessarily occur one immediately after the other, but all store accesses by other CPUs and channel programs and the fetch and store accesses associated with interlocked-update references by other CPUs are prevented from occurring at the same location between the fetch and the store accesses of an interlocked update reference.

A multi-processor system might incorporate various means to interlock storage operand references. One embodiment would have the processor obtaining exclusive ownership of the cache line or lines in the system during the references. Another embodiment would require that the storage accesses are restricted to the same cache line, for example by requiring that the operands being accessed from memory are on an integral boundary that would be within

a Cache line. In this case, any 64 bit (8 byte) operand being accessed in a 128 byte cache line is certainly wholly within the cache line if it is on an integral 64 bit boundary.

BLOCK CONCURRENT REFERENCES:

For some references, the accesses to all bytes (8 bits) within a halfword (2 bytes), word (4 bytes), doubleword (8 bytes), or quadword (16 bytes) are specified to appear to be block concurrent as observed by other CPUs and channel programs. The halfword, word, doubleword, or quadword is referred to in this section as a block. When a fetch-type reference is specified to appear to be concurrent within a block, no store access to the block by another CPU or channel program is permitted during the time that bytes contained in the block are being fetched. When a store-type reference is specified to appear to be concurrent within a block, no access to the block, either fetch or store, is permitted by another CPU or channel program during the time that the bytes within the block are being stored.

The term serializing instruction refers to an instruction which causes one or more serialization functions to be performed. The term serializing operation refers to a unit of operation within an instruction or to a machine operation such as an interruption which causes a serialization function is performed.

SPECIFIC-OPERAND SERIALIZATION:

Certain instructions may cause specific-operand serialization to be performed for an operand of the instruction. As observed by other CPUs and by the channel subsystem, a specific-operand-serialization operation consists in completing all conceptually previous storage accesses by the CPU before a conceptually subsequent accesses to the specific storage operand of the instruction may occur. At the completion of an instruction causing specific-operand serialization, the instruction's store is completed as observed by other CPUs and channel programs. Specific-operand serialization is performed by the execution of the following instructions:

- ADD IMMEDIATE (ASI, AGSI) and ADD LOGICALWITH SIGNED IMMEDIATE, for the first operand, when the interlocked-access facility is installed and the first operand is aligned on a boundary which is integral to the size of the operand.

• LOAD AND ADD, LOAD AND ADD LOGICAL, LOAD AND AND, LOAD AND EXCLUSIVE OR, LOAD AND OR, for the second operand.

INTERLOCKED UPDATE:

IBM z/Architecture and its predecessor multiprocessor architectures (dating back to later System 360s) have implemented certain "interlocked-update" instructions. An interlocked-update instruction ensures that the CPU on which the instruction executes has exclusive access to a memory location from the time the memory is fetched until it is stored back. This guarantees that multiple CPUs of a multi-processor configuration, attempting to access the same location will not observe erroneous results.

The first interlocked-update instruction was TEST AND SET (TS), introduced in S/360 multiprocessing systems. System 370 introduced the COMPARE AND SWAP (CS) and COMPARE DOUBLE AND SWAP (CDS) instructions. ESA/390 added the COMPARE AND SWAP AND PURGE (CSP) instruction (a specialized form used in virtual memory management). z/Architecture added the 64-bit COMPARE AND SWAP (CSG) and COMPARE AND SWAP AND PURGE (CSPG), and the 128-bit COMPARE DOUBLE AND SWAP (CDSG) instructions. The z/Architecture long-displacement facility added the COMPARE AND SWAP (CSY) and COMPARE DOUBLE AND SWAP (CDSY) instructions. The z/Architecture compare-and-swap-and-store facility added the COMPARE AND SWAP AND STORE instruction. Mnemonics such as (TS) for the TEST AND SET instruction are used by assembler programmers to identify the instruction. The assembler notation is discussed in the z/Architecture reference and is not significant to the teaching of the present invention.

By using the prior art interlocked-update instructions, more elaborate forms of serialized access can be effected, including locking protocols, interlocked arithmetic and logical operations to memory locations, and much more, but at a cost of complexity and additional CPU cycles. There is a persistent need for a wider variety of interlocked-update paradigms that operate as an atomic unit of operation. Embodiments herein address three of these paradigms.

This disclosure describes two new sets of instructions that implement interlocked update techniques, and enhancements to a third set of existing instructions that are defined to operate using interlocked update when the operands are appropriately aligned:

Load and Perform Operation:

This group of instructions loads a value from a memory location (the second operand) into a general register (the first operand), performs an arithmetic or boolean operation on the value in a general register (the third operand), and places the result of the operation back into the memory location. The fetch and store of the second operand appears to be a block-concurrent interlocked update to other CPUs.

Load Pair Disjoint:

This group of instructions attempts to load two values from distinct, separate memory locations (the first and second operands) into an even/odd pair of general registers (designated as the third operand). Whether or not the two distinct memory locations are accessed in an interlocked manner (that is, without one of the values being changed by another CPU) is indicated by the condition code.

ADD [LOGICAL WITH SIGNED] IMMEDIATE Enhancements:

The prior art System z10 introduced several instructions to perform addition to memory locations using an immediate constant in the instruction: ADD IMMEDIATE (ASI, AGSI) and ADD LOGICAL WITH SIGNED IMMEDIATE (ALSI, ALGSI). As originally defined, the memory accesses by these instructions were not interlocked update. When the interlocked-update facility is installed and the memory operand for these instructions is aligned on an integral boundary, the fetch/addition/store of the operand is now defined to be a block-concurrent interlocked update.

Other architectures implement alternative solutions to this problem. For example, the Intel Pentium architecture defines a LOCK prefix instruction that affects interlocked-update for certain subsequent instructions. However, the locking-prefix technique adds complexity to the architecture that is unnecessary. The solution described herein effects interlocked update in an atomic unit of operation -- without the need for a prefix instruction.

INTERLOCKED-STORAGE-ACCESS INSTRUCTIONS:

The following are examples of Interlocked-Storage Access instructions.

LOAD AND ADD (RSY FORMAT)

When the instruction is executed by the computer system, the second operand is added to the third operand, and the sum is placed at the second-operand location. Subsequently, the original contents of the second operand (prior to the addition) are placed unchanged at the first-operand location. For LAA OpCode, the operands are treated as being 32-bit signed binary integers. For LAAG OpCode, the operands are treated as being 64-bit signed binary integers. The fetch of the second operand for purposes of loading and the store into the second-operand location appear to be a block-concurrent interlocked update reference as observed by other CPUs. A specific-operand-serialization operation is performed. The displacement is treated as a 20-bit signed binary integer. The second operand of LAA must be designated on a word boundary. The second operand of LAAG must be designated on a doubleword boundary. Otherwise, a specification exception is generated.

Resulting Condition Code:

- 0 Result zero; no overflow
- 1 Result less than zero; no overflow
- 2 Result greater than zero; no overflow
- 3 Overflow

Program Exceptions:

- Access (fetch and store, operand 2)
- Fixed-point overflow
- Operation (if the interlocked-access facility is not installed)
- Specification

Programming Notes:

1. Except for the case where the R1 and R3 fields designate the same register, general register R3 is unchanged.

2. The operation of LOAD AND ADD, LOAD ANDADD LOGICAL, LOAD AND AND, LOAD ANDEXCLUSIVE OR, and LOAD AND OR may be expressed as follows.

$\text{temp} \leftarrow \text{operand_2}; \text{operand_2} \leftarrow \text{operand_2 OP operand_3}; \text{operand_1} \leftarrow \text{temp};$ OP represents the arithmetic or logical operation being performed by the instruction.

LOAD AND ADD LOGICAL (RSY FORMAT)

When the instruction is executed by the computer system, the second operand is added to the third operand, and the sum is placed at the second-operand location. Subsequently, the original contents of the second operand (prior to the addition) are placed unchanged at the first-operand location. For LAAL OpCode, the operands are treated as being 32-bit unsigned binary integers. For LAALG OpCode, the operands are treated as being 64-bit unsigned binary integers. The fetch of the second operand for purposes of loading and the store into the second-operand location appear to be a block-concurrent interlocked update reference as observed by other CPUs. A specific-operand-serialization operation is performed. The displacement is treated as a 20-bit signed binary integer. The second operand of LAAL must be designated on a word boundary. The second operand of LAALG must be designated on a doubleword boundary. Otherwise, a specification exception is generated.

Resulting Condition Code:

- 0 Result zero; no carry
- 1 Result not zero; no carry
- 2 Result zero; carry
- 3 Result not zero; carry

Program Exceptions:

- Access (fetch and store, operand 2)
- Operation (if the interlocked-access facility is not installed)
- Specification

Programming Note: See the programming notes for LOAD AND ADD.

LOAD AND AND (RSY FORMAT)

When the instruction is executed by the computer system, the AND of the second operand and third operand is placed at the second-operand location. Subsequently, the original contents of the second operand (prior to the AND operation) are placed unchanged at the first-operand location. For LAN OpCode, the operands are 32 bits. For LANG OpCode, the operands are 64 bits. The connective AND is applied to the operands bit by bit. The contents of a bit position in the result are set to one if the corresponding bit positions in both operands contain ones; otherwise, the result bit is set to zero. The fetch of the second operand for purposes of loading and the store into the second-operand location appear to be a block-concurrent interlocked update reference as observed by other CPUs. A specific-operand-serialization operation is performed. The displacement is treated as a 20-bit signed binary integer. The second operand of LAN must be designated on a word boundary. The second operand of LANG must be designated on a doubleword boundary. Otherwise, a specification exception is generated.

Resulting Condition Code:

0 Result zero

1 Result not zero

2 --

3 --

Program Exceptions:

- Access (fetch and store, operand 2)
- Operation (if the interlocked-access facility is not installed)
- Specification

Programming Note: See the programming notes for LOAD AND ADD.

LOAD AND EXCLUSIVE OR (RSY FORMAT)

When the instruction is executed by the computer system, the EXCLUSIVE OR of the second operand and third operand is placed at the second-operand location. Subsequently, the original contents of the second operand (prior to the EXCLUSIVE OR operation) are placed unchanged at the first-operand location. For LAX OpCode, the operands are 32 bits.

For LAXG OpCode, the operands are 64 bits. The connective exclusive OR is applied to the operands bit by bit. The contents of a bit position in the result are set to one if the bits in the corresponding bit positions in the two operands are unlike; otherwise, the result bit is set to zero. The fetch of the second operand for purposes of loading and the store into the second-operand location appear to be a block-concurrent interlocked update reference as observed by other CPUs. A specific-operand-serialization operation is performed. The displacement is treated as a 20-bit signed binary integer. The second operand of LAX must be designated on a word boundary. The second operand of LAXG must be designated on a doubleword boundary. Otherwise, a specification exception is generated.

Resulting Condition Code:

- 0 Result zero
- 1 Result not zero
- 2 --
- 3 --

Program Exceptions:

- Access (fetch and store, operand 2)
- Operation (if the interlocked-access facility is not installed)
- Specification

Programming Note: See the programming notes for LOAD AND ADD.

LOAD AND OR (RSY FORMAT)

When the instruction is executed by the computer system, the OR of the second operand and third operand is placed at the second-operand location. Subsequently, the original contents of the second operand (prior to the OR operation) are placed unchanged at the first-operand location. For LAO OpCode, the operands are 32 bits. For LAOG OpCode, the operands are 64 bits. The connective OR is applied to the operands bit by bit. The contents of a bit position in the result are set to one if the corresponding bit position in one or both operands contains a one; otherwise, the result bit is set to zero. The fetch of the second operand for purposes of loading and the store into the second-operand location appear to be a block-

concurrent interlocked update reference as observed by other CPUs. A specific-operand-serialization operation is performed. The displacement is treated as a 20-bit signed binary integer. The second operand of LAO must be designated on a word boundary. The second operand of LAOG must be designated on a doubleword boundary. Otherwise, a specification exception is generated.

Resulting Condition Code:

0 Result zero

1 Result not zero

2 --

3 --

Program Exceptions:

- Access (fetch and store, operand 2)
- Operation (if the interlocked-access facility is not installed)
- Specification

Programming Note: See the programming notes for LOAD AND ADD.

LOAD PAIR DISJOINT (SSF FORMAT)

When the instruction is executed by the computer system, the General register R3 designates the even numbered register of an even/odd register pair. The first operand is placed unchanged into the even numbered register of the third operand, and the second operand is placed unchanged into odd-numbered register of the third operand. The condition code indicates whether the first and second operands appear to be fetched by means of block-concurrent interlocked fetch. For LPD OpCode, the first and second operands are words in storage, and the third operand is in bits 32-63 of general registers R3 and R₃₊₁; bits 0-31 of the registers are unchanged. For LPDG OpCode, the first and second operands are doublewords in storage, and the third operand is in bits 0-63 of general registers R3 and R₃₊₁. When, as observed by other CPUs, the first and second operands appear to be fetched by means of block-concurrent interlocked fetch, condition code 0 is set. When the first and second operands do not appear to be fetched by means of block-concurrent interlocked

update, condition code 3 is set. The third operand is loaded regardless of the condition code. The displacement of the first and second operands is treated as a 12-bit unsigned binary integer. The first and second operands of LPD must be designated on a word boundary. The first and second operands of LPDG must be designated on a doubleword boundary. General register R3 must designate the even numbered register. Otherwise, a specification exception is generated.

Resulting Condition Code:

- 0 Register pair loaded by means of interlocked fetch
- 1 --
- 2 --
- 3 Register pair not loaded by means of interlocked fetch

Program Exceptions:

- Access (fetch, operands 1 and 2)
- Operation (if the interlocked-access facility is not installed)
- Specification

Programming Notes:

1. The setting of the condition code is dependent upon storage accesses by other CPUs in the configuration.
2. When the resulting condition code is 3, the program may branch back to re-execute the LOADPAIR DISJOINT instruction. However, after repeated unsuccessful attempts to attain an interlocked fetch, the program should use an alternate means of serializing access to the storage operands. It is recommended that the program re-execute the LOAD PAIR DISJOINT no more than 10 times before branching to the alternate path.
3. The program should be able to accommodate a situation where condition code 0 is never set.

LOAD/STORE-ON-CONDITION INSTRUCTIONS:

The following are example Load/Store-on-condition instructions:

LOAD ON CONDITION (RRF, RSY FORMAT)

When the instruction is executed by the computer system, the second operand is placed unchanged at the first operand location if the condition code has one of the values specified by M3; otherwise, the first operand remains unchanged. For LOC and LROC, the first and second operands are 32 bits, and for LGOC OpCode and LGROC OpCode, the first and second operands are 64 bits. The M3 field is used as a four-bit mask. The four condition codes (0, 1, 2, and 3) correspond, left to right, with the four bits of the mask, as follows:

The current condition code is used to select the corresponding mask bit. If the mask bit selected by the condition code is one, the load is performed. If the mask bit selected is zero, the load is not performed. The displacement for LOC and LGOC is treated as a 20-bit signed binary integer. For LOC and LGOC, when the condition specified by the M3 field is not met (that is, the load operation is not performed), it is model dependent whether an access exception, or PER zero-address detection is generated for the second operand.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2 of LOC and LGOC)
- Operation (if the load/store-on-condition facility is not installed)

Programming Notes:

1. When the M3 field contain zeros, the instruction acts as a NOP. When the M3 field contains all ones and no exception condition exists, the load operation is always performed. However, these are not the preferred means of implementing a NOP or unconditional load, respectively.
2. For LOC and LGOC, when the condition specified by the M3 field is not met, it is model dependent whether the second operand is brought into the cache.
3. LOAD ON CONDITION provides a function similar to that of a separate BRANCH ON CONDITION instruction followed by a LOAD instruction, except that LOAD ON CONDITION does not provide an index register. For example, the following two instruction sequences are equivalent. On models that implement predictive branching, the combination of the BRANCH ON CONDITION and LOAD instructions may perform somewhat better

than the LOAD ON CONDITION instruction when the CPU is able to successfully predict the branch condition. However, on models where the CPU is not able to successfully predict the branch condition, such as when the condition is more random, the LOAD ON CONDITION instruction may provide significant performance improvement.

STORE ON CONDITION (RSY FORMAT)

When the instruction is executed by the computer system, the first operand is placed unchanged at the second operand location if the condition code has one of the values specified by M3; otherwise, the second operand remains unchanged. For STOC OpCode, the first and second operands are 32 bits, and for STGOC OpCode, the first and second operands are 64 bits. The M3 field is used as a four-bit mask. The four condition codes (0, 1, 2, and 3) correspond, left to right, with the four bits of the mask, as follows: The current condition code is used to select the corresponding mask bit. If the mask bit selected by the condition code is one, the store is performed. If the mask bit selected is zero, the store is not performed. normal instruction sequencing proceeds with the next sequential instruction. The displacement is treated as a 20-bit signed binary integer. When the condition specified by the M3 field is not met (that is, store operation is not performed), it is model dependent whether any or all of the following occur for the second operand: (a) an access exception is generated, (b) a PER storage-alteration event is generated, (c) a PER zero-address-detection event is generated, or (d) the change bit is set.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (store, operand 2)
- Operation (if the load/store-on-condition facility is not installed)

Programming Notes:

1. When the M3 field contain zeros, the instruction acts as a NOP. When the M3 field contains all ones and no exception condition exists, the store operation is always performed. However, these are not the preferred means of implementing a NOP or unconditional store, respectively.

2. When the condition specified by the M3 field is not met, it is model dependent whether the second operand is brought into the cache.
3. STORE ON CONDITION provides a function similar to that of a separate BRANCH ON CONDITION instruction followed by a STORE instruction, except that STORE ON CONDITION does not provide an index register. For example, the following two instruction sequences are equivalent. On models that implement predictive branching, the combination of the BRANCH ON CONDITION and STORE instructions may perform somewhat better than the STORE ON CONDITION instruction when the CPU is able to successfully predict the branch condition. However, on models where the CPU is not able to successfully predict the branch condition, such as when the condition is more random, the STORE ONCONDITION instruction may provide significant performance improvement.

DISTINCT-OPERANDS-FACILITY INSTRUCTIONS:

The following are example Distinct-operand-facility instructions:

ADD (RR, RRE, RRF, RX, RXY FORMAT), ADD IMMEDIATE (RIL, RIE, SIY FORMAT)

When the instruction is executed by the computer system, for ADD (A, AG, AGF, AGFR, AGR, AR, and AY OpCodes) and for ADD IMMEDIATE (AFI, AGFI, AGSI, and ASI OpCodes), the second operand is added to the first operand, and the sum is placed at the first-operand location. For ADD (AGRK and ARK) and for ADD IMMEDIATE (AGHIK and AHIK OpCodes), the second operand is added to the third operand, and the sum is placed at the first operand location.

For ADD (A, AR, ARK, and AY OpCodes) and for ADD IMMEDIATE (AFI OpCodes), the operands and the sum are treated as 32-bit signed binary integers. For ADD (AG, AGR, and AGRK OpCodes), they are treated as 64-bit signed binary integers.

For ADD (AGFR, AGF OpCodes) and for ADD IMMEDIATE (AGFI OpCode), the second operand is treated as a 32-bit signed binary integer, and the first operand and the sum are treated as 64-bit signed binary integers. For ADD IMMEDIATE (ASI OpCode), the second operand is treated as an 8-bit signed binary integer, and the first operand and the sum are

treated as 32-bit signed binary integers. For ADD IMMEDIATE (AGSI OpCode), the second operand is treated as an 8-bit signed binary integer, and the first operand and the sum are treated as 64-bit signed binary integers. For ADD IMMEDIATE (AHIK OpCode), the first and third operands are treated as 32-bit signed binary integers, and the second operand is treated as a 16-bit signed binary integer. For ADD IMMEDIATE (AGHIK OpCode), the first and third operands are treated as 64-bit signed binary integers, and the second operand is treated as a 16-bit signed binary integer.

When there is an overflow, the result is obtained by allowing any carry into the sign-bit position and ignoring any carry out of the sign-bit position, and condition code 3 is set. If the fixed-point-overflow mask is one, a program interruption for fixed-point overflow occurs.

When the interlocked-access facility is installed and the first operand of ADD IMMEDIATE (ASI, AGSI) is aligned on an integral boundary corresponding to its size, then the fetch and store of the first operand are performed as an interlocked update as observed by other CPUs, and a specific-operand-serialization operation is performed. When the interlocked-access facility is not installed, or when the first operand of ADD IMMEDIATE (ASI, AGSI) is not aligned on an integral boundary corresponding to its size, then the fetch and store of the operand are not performed as an interlocked update.

The displacement for A is treated as a 12-bit unsigned binary integer. The displacement for AY, AG, AGF, AGSI and ASI, is treated as a 20-bit signed binary integer.

Resulting Condition Code:

- 0 Result zero; no overflow
- 1 Result less than zero; no overflow
- 2 Result greater than zero; no overflow
- 3 Overflow

Program Exceptions:

- * Access (fetch and store, operand 1 of AGSI and ASI only; fetch, operand 2 of A, AY, AG, and AGF only)

- Fixed-point overflow
- Operation (AY, if the long-displacement facility is not installed; AFI and AGFI, if the extendedimmediate facility is not installed; AGSI and ASI, if the general-instructions-extension facility is not installed; ARK, AGRK, AHIK, and AGHIK, if the distinct-operands facility is not installed)

Programming Notes:

1. Accesses to the first operand of ADD IMMEDIATE (AGSI and ASI) consist in fetching a firstoperand from storage and subsequently storing the updated value. When the interlocked-access facility is not installed, or when the first operand is not aligned on an integral boundary corresponding to its size, the fetch and store accesses to the first operand do not necessarily occur one immediately after the other. Under such conditions, ADD IMMEDIATE (AGSI and ASI) cannot be safely used to update a location in storage if the possibility exists that another CPU or the channel subsystem may also be updating the location. When the interlocked-access facility is installed and the first operand is aligned on an integral boundary corresponding to its size, the operand is accessed using a block-concurrent interlocked update.
2. For certain programming languages which ignore overflow conditions on arithmetic operations, the setting of condition code 3 obscures the sign of the result. However, for ADD IMMEDIATE, the sign of the I2 field (which is known at the time of code generation) may be used in setting a branch mask which will accurately determine the resulting sign.

ADD LOGICAL (RR, RRE, RX, RXY Format) ADD LOGICAL IMMEDIATE (RIL Format)

When the instruction is executed by the computer system, for ADD LOGICAL (AL, ALG, ALGF, ALGFR, ALGR, ALR, and ALY OpCodes) and for ADD LOGICAL IMMEDIATE (ALGFI and ALFI OpCodes), the second operand is added to the first operand, and the sum is placed at the firstoperand location.

For ADD LOGICAL (ALGRK and ALRK OpCodes), the second operand is added to the third operand, and the sum is placed at the first-operand location. For ADD LOGICAL (AL, ALR, ALRK, and ALY OpCodes) and for ADD LOGICAL IMMEDIATE (ALFI OpCode),

the operands and the sum are treated as 32-bit unsigned binary integers. For ADD LOGICAL (ALG, ALGR, and ALGRK OpCodes), they are treated as 64-bit unsigned binary integers. For ADD LOGICAL (ALGFR, ALGF OpCodes) and for ADD LOGICAL IMMEDIATE (ALGFI OpCode), the second operand is treated as a 32-bit unsigned binary integer, and the first operand and the sum are treated as 64-bit unsigned binary integers.

The displacement for AL is treated as a 12-bit unsigned binary integer. The displacement for ALY, ALG, and ALGF is treated as a 20-bit signed binary integer.

Resulting Condition Code:

0 Result zero; no carry

1 Result not zero; no carry 2 Result zero; carry

3 Result not zero; carry

Program Exceptions:

- Access (fetch, operand 2 of AL, ALY, ALG, and ALGF only)
- Operation (ALY, if the long-displacement facility is not installed; ALFI and ALGFI, if the extended immediate facility is not installed; ALRK and ALGRK, if the distinct-operands facility is not installed)

ADD LOGICAL WITH SIGNED IMMEDIATE (SIY, RIE Format)

When the instruction is executed by the computer system, for ALGSI OpCode and ALSI OpCode, the second operand is added to the first operand, and the sum is placed at the first operand location. For ALGHSIK and ALHSIK, OpCodes the second operand is added to the third operand, and the sum is placed at the first-operand location. For ALSI OpCode, the first operand and the sum are treated as 32-bit unsigned binary integers. For ALGSI OpCodes, the first operand and the sum are treated as 64-bit unsigned binary integers. For both ALSI and ALGSI, the second operand is treated as an 8-bit signed binary integer. For ALHSIK OpCode, the first and third operands are treated as 32-bit unsigned binary integers. For ALGHSIK OpCode, the first and third operands are treated as 64-bit unsigned binary integers. For both ALGHSIK and ALHSIK, the second operand is treated as a 16-bit signed binary integer.

When the interlocked-access facility is installed and the first operand is aligned on an integral boundary corresponding to its size, the operand is accessed using a block-concurrent interlocked update. For ALGSI and ALSI, the second operand is added to the first operand, and the sum is placed at the first operand location. For ALGHSIK and ALHSIK, the second operand is added to the third operand, and the sum is placed at the first-operand location. For ALSI, the first operand and the sum are treated as 32-bit unsigned binary integers. For ALGSI, the first operand and the sum are treated as 64-bit unsigned binary integers. For both ALSI and ALGSI, the second operand is treated as an 8-bit signed binary integer. For ALHSIK, the first and third operands are treated as 32-bit unsigned binary integers. For ALGHSIK, the first and third operands are treated as 64-bit unsigned binary integers. For both ALGHSIK and ALHSIK, the second operand is treated as a 16-bit signed binary integer. When the interlocked-access facility is installed and the first operand is aligned on an integral boundary corresponding to its size, then the fetch and store of the first operand is performed as an interlocked update as observed by other CPUs, and a specific operand-serialization operation is performed. When the interlocked-access facility is not installed, or when the first operand of ADD LOGICAL WITH SIGNED IMMEDIATE (ALSI, ALGSI) is not aligned on an integral boundary corresponding to its size, then the fetch and store of the operand are not performed as an interlocked update. When the second operand contains a negative value, the condition code is set as though a SUBTRACT LOGICAL operation was performed. Condition code 0 is never set when the second operand is negative. The displacement is treated as a 20-bit signed binary integer.

Resulting Condition Code:

- 0 Result zero; no carry
- 1 Result not zero; no carry
- 2 Result zero; carry
- 3 Result not zero; carry

AND (RR, RRE, RRF, RX, RXY, SI, SIY, SS FORMAT)

When the instruction is executed by the computer system, for N, NC, NG, NGR, NI, NIY, NR, and NY OpCodes, the AND of the first and second operands is placed at the first operand location. For NGRK and NRK, the AND of the second and third operands is placed

at the first operand location. The connective AND is applied to the operands bit by bit. The contents of a bit position in the result are set to one if the corresponding bit positions in both operands contain ones; otherwise, the result bit is set to zero. For AND (NC OpCode), each operand is processed left to right. When the operands overlap, the result is obtained as if the operands were processed one byte at a time and each result byte were stored immediately after fetching the necessary operand bytes. For AND (NI and NIY OpCodes), the first operand is one byte in length, and only one byte is stored. For AND (N, NR, NRK, and NY), the operands are 32bits, and for AND (NG, NGR, and NGRK OpCodes), they are 64 bits. The displacements for N, NI, and both operands of NC are treated as 12-bit unsigned binary integers. The displacement for NY, NIY, and NG is treated as a 20-bit signed binary integer.

Resulting Condition Code:

0 Result zero

1 Result not zero

2 --

3 --

Program Exceptions:

- Access (fetch, operand 2, N, NY, NG, and NC; fetch and store, operand 1, NI, NIY, and NC)
- Operation (NIY and NY, if the long-displacement facility is not installed; NGRK and NRK, if the distinct operands facility is not installed)

EXCLUSIVE OR (RR, RRE, RRF, RX, RXY, SI, SIY, SS FORMAT)

When the instruction is executed by the computer system, for X, XC, XG, XGR, XI, XIY, XR, and XY OpCodes, the EXCLUSIVE OR of the first and second operands is placed at the first-operand location. For XGRK and XRK OpCodes, the EXCLUSIVE OR of the second and third operands is placed at the first-operand location. The connective EXCLUSIVE OR is applied to the operands bit by bit. The contents of a bit position in the result are set to one if the bits in the corresponding bit positions in the two operands are unlike; otherwise, the result bit is set to zero. For EXCLUSIVE OR (XC OpCodes), each operand is processed left to right. When the operands overlap, the result is obtained as if the

operands were processed one byte at a time and each result byte were stored immediately after fetching the necessary operand bytes. For EXCLUSIVE OR (XI, XIY OpCodws), the first operand is one byte in length, and only one byte is stored. For EXCLUSIVE OR (X, XR, XRK, and XY OpCodws), the operands are 32 bits, and for EXCLUSIVE OR (XG,XGR, and XGRK OpCodws), they are 64 bits. The displacements for X, XI, and both operands of XC are treated as 12-bit unsigned binary integers. The displacement for XY, XIY, and XG is treated as a 20-bit signed binary integer.

Resulting Condition Code:

0 Result zero

1 Result not zero

2 --

3 --

Program Exceptions:

- Access (fetch, operand 2, X, XY, XG, and XC; fetch and store, operand 1, XI, XIY, and XC)
- Operation (XIY and XY, if the long-displacement facility is not installed; XGRK and XRK, if the distinct operands facility is not installed)

Programming Notes:

1. -
2. EXCLUSIVE OR may be used to invert a bit, an operation particularly useful in testing and setting programmed binary switches.
3. A field EXCLUSIVE-ORed with itself becomes allzeros.
4. For EXCLUSIVE OR (XR or XGR), the sequence A EXCLUSIVE-OR B, B EXCLUSIVE-OR A, A EXCLUSIVE-OR B results in the exchange of the contents of A and B without the use of an additional general register.
5. Accesses to the first operand of EXCLUSIVE OR (XI) and EXCLUSIVE OR (XC) consist in fetching a first-operand byte from storage and subsequently storing the updated value. These fetch and store accesses to a particular byte do not necessarily occur one immediately after the other. Thus, EXCLUSIVE OR cannot be safely used to update a location in storage if the possibility exists that another CPU or a channel program may also be updating the location.

OR (RR, RRE, RRF, RX, RXY, SI, SIY, SS FORMAT)

When the instruction is executed by the computer system, for O, OC, OG, OGR, OI, OIY, OR, and OY OpCodes, the OR of the first and second operands is placed at the first operand location. For OGRK and ORK, the OR of the second and third operands is placed at the first-operand location. The connective OR is applied to the operands bit by bit. The contents of a bit position in the result are set to one if the corresponding bit position in one or both operands contains a one; otherwise, the result bit is set to zero. For OR (OC OpCode), each operand is processed left to right. When the operands overlap, the result is obtained as if the operands were processed one byte at a time and each result byte were stored immediately after fetching the necessary operand bytes. For OR (OI, OIY OpCodes), the first operand is one byte in length, and only one byte is stored. For OR (O, OR, ORK, and OY OpCodes), the operands are 32bits, and for OR (OG, OGR, and OGRK OpCodes), they are 64bits. The displacements for O, OI, and both operands of OC are treated as 12-bit unsigned binary integers. The displacement for OY, OIY, and OG is treated as a 20-bit signed binary integer.

Resulting Condition Code:

0 Result zero

1 Result not zero

2 --

3 --

SHIFT LEFT SINGLE (RS, RSY FORMAT)

When the instruction is executed by the computer system, for SLA OpCode, the 31-bit numeric part of the signed first operand is shifted left the number of bits specified by the second-operand address, and the result is placed at the first-operand location. Bits 0-31 of general register R1 remain unchanged. For SLAK OpCode, the 31-bit numeric part of the signed third operand is shifted left the number of bits specified by the second-operand address, and the result, with the sign bit of the third operand appended on its left, is placed at the first-operand location. Bits 0-31 of general register R1 remain unchanged, and the third operand remains unchanged in general register R3. For SLAG OpCode, the 63-bit numeric part of the signed third operand is shifted left the number of bits specified by the second-operand address, and the result, with the sign bit of the third operand appended on its left, is

placed at the first-operand location. The third operand remains unchanged in general register R3. The second-operand address is not used to address data; its rightmost six bits indicate the number of bit positions to be shifted. The remainder of the address is ignored. For SLA OpCode, the first operand is treated as a 32-bit signed binary integer in bit positions 32-63 of general register R1. The sign of the first operand remains unchanged. All 31 numeric bits of the operand participate in the left shift. For SLAK, the first and third operands are treated as 32-bit signed binary integers in bit positions 32-63 of general registers R1 and R3, respectively. The sign of the first operand is set equal to the sign of the third operand. All 31 numeric bits of the third operand participate in the left shift. For SLAG, the first and third operands are treated as 64-bit signed binary integers in bit positions 0-63 of general registers R1 and R3, respectively. The sign of the first operand is set equal to the sign of the third operand. All 63 numeric bits of the third operand participate in the left shift. For SLA, SLAG, or SLAK, zeros are supplied to the vacated bit positions on the right. If one or more bits unlike the sign bit are shifted out of bit position 33, for SLA or SLAK, or bit position 1, for SLAG, an overflow occurs, and condition code 3 is set. If the fixed-point-overflow mask bit is one, a program interruption for fixed-point overflow occurs.

Resulting Condition Code:

0 Result zero; no overflow

1 Result less than zero; no overflow

2 Result greater than zero; no overflow

3 Overflow•

Fixed-point overflow

• Operation (SLAK, if the distinct-operands facility is not installed)

SHIFT LEFT SINGLE LOGICAL (RS, RSY FORMAT)

When the instruction is executed by the computer system, for SLL OpCode, the 32-bit first operand is shifted left the number of bits specified by the second-operand address, and the result is placed at the first-operand location. Bits 0-31 of general register R1 remain unchanged. For SLLK, the 32-bit third operand is shifted left the number of bits specified by the second-operand address, and the result is placed at the first-operand location. Bits 0-31 of general register R1 remain unchanged, and the third operand remains unchanged in

general register R3. For SLLG OpCode, the 64-bit third operand is shifted left the number of bits specified by the second-operand address, and the result is placed at the first-operand location. The third operand remains unchanged in general register R3. The second-operand address is not used to address data; its rightmost six bits indicate the number of bit positions to be shifted. The remainder of the address is ignored. For SLL, the first operand is in bit positions 32-63 of general register R1. All 32 bits of the operand participate in the left shift. For SLLK, the first and third operands are in bit positions 32-63 of general registers R1 and R3, respectively. All 32 bits of the third operand participate in the left shift. For SLLG, the first and third operands are in bit positions 0-63 of general registers R1 and R3, respectively. All 64 bits of the third operand participate in the left shift. For SLL, SLLG, or SLLK OpCodes, zeros are supplied to the vacated bit positions on the right.

Condition Code: The code remains unchanged.

Program Exceptions:

- Operation (SLLK, if the distinct-operands facility is not installed)

SHIFT RIGHT SINGLE (RS, RSY FORMAT)

When the instruction is executed by the computer system, for SRA OpCode, the 31-bit numeric part of the signed first operand is shifted right the number of bits specified by the second-operand address, and the result is placed at the first-operand location. Bits 0-32 of general register R1 remain unchanged. For SRAK OpCode, the 31-bit numeric part of the signed third operand is shifted right the number of bits specified by the second-operand address, and the result, with the sign bit of the third operand appended on its left, is placed at the first-operand location. Bits 0-32 of general register R1 remain unchanged. For SHIFT RIGHT SINGLE (SRAG OpCode), the 63-bit numeric part of the signed third operand is shifted right the number of bits specified by the second-operand address, and the result, with the sign bit of the third operand appended on its left, is placed at the first-operand location. The third operand remains unchanged in general register R3. The second-operand address is not used to address data; its rightmost six bits indicate the number of bit positions to be shifted. The remainder of the address is ignored. For SRA, The first operand is treated as a 32-bit signed binary integer in bit positions 32-63 of general register R1. The sign of the first

operand remains unchanged. All 31 numeric bits of the operand participate in the right shift. For SRAK, the first and third operands are treated as 32-bit signed binary integers in bit positions 32-63 of general registers R1 and R3, respectively. The sign of the first operand is set equal to the sign of the third operand. All 31 numeric bits of the third operand participate in the right shift. For SRAG, the first and third operands are treated as 64-bit signed binary integers in bit positions 0-63 of general registers R1 and R3, respectively. The sign of the first operand is set equal to the sign of the third operand. All 63 numeric bits of the third operand participate in the right shift. For SRA, SRAG, or SRAK, bits shifted out of bit position 63 are not inspected and are lost. Bits equal to the sign are supplied to the vacated bit positions on the left.

Resulting Condition Code:

0 Result zero

1 Result less than zero

2 Result greater than zero

3 --

Program Exceptions:

- Operation (SRAK, if the distinct-operands facility is not installed)

Programming Notes:

1. A right shift of one bit position is equivalent to division by 2 with rounding downward. When an even number is shifted right one position, the result is equivalent to dividing the number by 2. When an odd number is shifted right one position, the result is equivalent to dividing the next lower number by 2. For example, +5 shifted right by one bit position yields +2, whereas -5 yields -3.
2. For SHIFT RIGHT SINGLE (SRA and SRAK), shift amounts from 31 to 63 cause the entire numeric part to be shifted out of the register, leaving a result of -1 or zero, depending on whether or not the initial contents were negative. For SHIFT RIGHT SINGLE (SRAG), a shift amount of 63 causes the same effect.

SHIFT RIGHT SINGLE LOGICAL (RS, RSY FORMAT)

When the instruction is executed by the computer system, for SRL OpCode,, the 32-bit first operand is shifted right the number of bits specified by the second-operand address, and the result is placed at the first-operand location. Bits 0-31 of general register R1 remain unchanged. For SRLK OpCode,, the 32-bit third operand is shifted right the number of bits specified by the second-operand address, and the result is placed at the first-operand location. Bits 0-31 of general register R1 remain unchanged, and the third operand remains unchanged in general register R3. For SRLG OpCode,, the 64-bit third operand is shifted right the number of bits specified by the second-operand address, and the result is placed at the first-operand location. The third operand remains unchanged in general register R3. The second-operand address is not used to address data; its rightmost six bits indicate the number of bit positions to be shifted. The remainder of the address is ignored. For SRL, the first operand is in bit positions 32-63 of general register R1. All 32 bits of the operand participate in the right shift. For SRLK, the first and third operands are in bit positions 32-63 of general registers R1 and R3, respectively. All 32 bits of the third operand participate in the right shift. For SRLG, the first and third operands are in bit positions 0-63 of general registers R1 and R3, respectively. All 64 bits of the third operand participate in the right shift. For SRL, SRLG, or SRLK, bits shifted out of bit position 63 are not inspected and are lost. Zeros are supplied to the vacated bit positions on the left.

Condition Code: The code remains unchanged.

Program Exceptions:

- Operation (SRLK, if the distinct-operands facility is not installed)

SUBTRACT (RR, RRE, RRF, RX, RXY FORMAT)

When the instruction is executed by the computer system, for S, SG, SGF, SGFR, SGR, SR, and SY, the second operand is subtracted from the first operand, and the difference is placed at the first-operand location. For SGRK and SRK, the third operand is subtracted from the second operand, and the difference is placed at the first-operand location. For S, SR, SRK, and SY, the operands and the difference are treated as 32-bit signed binary integers. For SG, SGR, and SGRK, they are treated as 64-bit signed binary integers. For SGFR and SGF, the second operand is treated as a 32-bit signed binary integer, and the first operand and the

difference are treated as 64-bit signed binary integers. When there is an overflow, the result is obtained by allowing any carry into the sign-bit position and ignoring any carry out of the sign-bit position, and condition code 3 is set. If the fixed-point-overflow mask is one, a program interruption for fixed-point overflow occurs. The displacement for S is treated as a 12-bit unsigned binary integer. The displacement for SY, SG, and SGF is treated as a 20-bit signed binary integer.

Resulting Condition Code:

- 0 Result zero; no overflow
- 1 Result less than zero; no overflow
- 2 Result greater than zero; no overflow
- 3 Overflow

Program Exceptions:

- Access (fetch, operand 2 of S, SY, SG, and SGF only)
- Fixed-point overflow
- Operation (SY, if the long-displacement facility is not installed; SRK, SGRK, if the distinct-operands facility is not installed)

Programming Notes:

1. For SR and SGR, when R1 and R2 designate the same register, subtracting is equivalent to clearing the register.
2. Subtracting a maximum negative number from itself gives a zero result and no overflow.

SUBTRACT LOGICAL (RR, RRE, RRF, RX, RXY FORMAT), SUBTRACT LOGICAL IMMEDIATE (RIL FORMAT)

When the instruction is executed by the computer system, for SUBTRACT LOGICAL (SL, SLG, SLGF, SLGFR, SLGR, SLR, SLY) and for SUBTRACT LOGICAL IMMEDIATE, the second operand is subtracted from the first operand, and the difference is placed at the first-operand location. For SUBTRACT LOGICAL (SLGRK and SLRK), the third operand is subtracted from the second operand, and the difference is placed at the first-operand location. For SUBTRACT LOGICAL (SL, SLR, SLRK, and SLY) and for SUBTRACT

LOGICAL IMMEDIATE(SLFI), the operands and the difference are treated as 32-bit unsigned binary integers. For SUBTRACTLOGICAL (SLG, SLGR, and SLGRK), they are treated as 64-bit unsigned binary integers. For SUBTRACTLOGICAL (SLGFR, SLGF) and for SUBTRACTLOGICAL IMMEDIATE (SLGFI), the second operand is treated as a 32-bit unsigned binary integer, and the first operand and the difference are treated as 64-bit unsigned binary integers. The displacement for SL is treated as a 12-bit unsigned binary integer. The displacement for SLY, SLG, and SLGF is treated as a 20-bit signed binary integer.

Resulting Condition Code:

0 --

1 Result not zero; borrow

2 Result zero; no borrow

3 Result not zero; no borrow

Program Exceptions:

- Access (fetch, operand 2 of SL, SLY, SLG, and SLGF only)
- Operation (SLY, if the long-displacement facility is not installed; SLFI and SLGFI, if the extended immediate facility is not installed; SLRK and SLGRK, if the distinct-operands facility is not installed)

Programming Notes:

1. Logical subtraction is performed by adding the one's complement of the second operand and a value of one to the first operand. The use of the one's complement and the value of one instead of the two's complement of the second operand results in a carry when the second operand is zero.
2. SUBTRACT LOGICAL differs from SUBTRACT only in the meaning of the condition code and in the absence of the interruption for overflow.
3. A zero difference is always accompanied by a carry out of bit position 0 for SLGR, SLGFR, SLG, and SLGF or bit position 32 for SLR, SL, and SLY, and, therefore, no borrow.
4. The condition-code setting for SUBTRACT LOGICAL can also be interpreted as indicating the presence or absence of a carry,

POPULATION COUNT INSTRUCTION:

The following is an example Population Count instruction:

POPULATION COUNT (RRE FORMAT)

When the instruction is executed by the computer system, a count of the number of one bits in each of the eight bytes of general register R2 is placed into the corresponding byte of general register R1. Each byte of general register R1 is an 8-bit binary integer in the range of 0-8.

Resulting Condition Code:

0 Result zero

1 Result not zero

2 --

3 --

• Operation (if the population-count facility is not installed)

Programming Notes:

1. The condition code is set based on all 64 bits of general register R1. 2. The total number of one bits in a general register can be computed as shown below. In this example, general register 15 contains the number of bits to be counted; the result containing the total number of one bits in general register 15 is placed in general register 8. (General register 9 is used as a work register and contains residual values on completion.)

2. If there is a high probability that the results of the POPCNT instruction are zero, the program may insert a conditional branch instruction may be inserted to skip the adding and shifting operations based on the condition code set by POPCNT.

3. Using techniques similar to that shown in programming note 2, the number of one bits in a word, halfword, or noncontiguous bytes of the second operand may be determined.

In an embodiment, referring to FIGs 6A and 6B, an arithmetic/logical instruction 608 is executed, wherein the instruction comprises an interlocked memory operand, the arithmetic/logical instruction comprising an opcode field (OP), a first register field (R1) specifying a first operand in a first register, a second register field (B2) specifying a second register the second register specifying location of a second operand in memory, and a third

register field (R3) specifying a third register, the execution of the arithmetic/logical instruction comprises: obtaining 601 by a processor, a second operand from a location in memory specified by the second register, the second operand consisting of a value (the value may be saved 607 in a temporary store in an embodiment); obtaining 602 a third operand from the third register; performing 603 an opcode defined arithmetic operation or a logical operation based on the obtained second operand and the obtained third operand to produce a result; storing 604 the produced result in the location in memory; and saving 605 the value of the obtained second operand in the first register, wherein the value is not changed by executing the instruction.

In an embodiment, a condition code is saved 606, the condition code indicating the result is zero or the result is not zero.

In an embodiment, the opcode defined arithmetic operation 652 is an arithmetic or logical ADD, and the opcode defined logical operation is any one of an AND, an EXCLUSIVE-OR, or an OR, and the execution comprises: responsive to the result of the logical operation being negative, saving the condition code indicating the result is negative; responsive to the result of the logical operation being positive, saving the condition code indicating the result is positive; and responsive to the result of the logical operation being an overflow, saving the condition code indicating the result is an overflow.

In an embodiment, operand size is specified by the opcode, wherein one or more first opcodes specify 32 bit operands and one or more second opcodes specify 64 bit operands.

In an embodiment, the arithmetic/logical instruction 608 further comprises the opcode consisting of two separate opcode fields (OP, OP), a first displacement field (DH2) and a second displacement field (DL2), wherein the location in memory is determined by adding contents of the second register to a signed displacement value, the signed displacement value comprising a sign extended value of the first displacement field concatenated to the second displacement field.

In an embodiment, the execution further comprises: responsive to the opcode being a first opcode and the second operand not being on a 32 bit boundary, generating a specification exception; and responsive to the opcode being a second opcode and the second operand not being on a 64 bit boundary, generating a specification exception.

In an embodiment, the processor is a processor in a multi-processor system, and the execution further comprises: the obtaining the second operand comprising preventing other processors of the multi-processor system from accessing the location in memory between said obtaining of the second operand and storing a result at the second location in memory; and upon said storing the produced result, permitting other processors of the multi-processor system to access the location in memory.

While the preferred embodiments have been illustrated and described herein, it is to be understood that the embodiments are not limited to the precise construction herein disclosed, and the right is reserved to all changes and modifications coming within the scope of the invention as defined in the appended claims.

CLAIMS

1. A computer implemented method for executing an arithmetic/logical instruction having an interlocked memory operand, the arithmetic/logical instruction comprising an opcode field, a first register field specifying a first operand in a first register, a second register field specifying a second register, the second register specifying location of a second operand in memory, and a third register field specifying a third register, the execution of the arithmetic/logical instruction comprising:

obtaining by a processor, a second operand from a location in memory specified by the second register, the second operand consisting of a value;

obtaining a third operand from the third register;

performing an opcode defined arithmetic operation or a logical operation based on the obtained second operand and the obtained third operand to produce a result;

storing the produced result in the location in memory; and

saving the value of the obtained second operand in the first register.

2. The method according to Claim 1, further comprising saving a condition code, the condition code indicating the result is zero or the result is not zero.

3. The method according to Claim 2, wherein the opcode defined arithmetic operation is an arithmetic or logical ADD, and wherein the opcode defined logical operation is any one of an AND, an EXCLUSIVE-OR, or an OR, further comprising:

responsive to the result of the logical operation being negative, saving the condition code indicating the result is negative;

responsive to the result of the logical operation being positive, saving the condition code indicating the result is positive; and

responsive to the result of the logical operation being an overflow, saving the condition code indicating the result is an overflow.

4. The method according to Claim 3, wherein the operand size is specified by the opcode, wherein one or more first opcodes specify 32 bit operands and one or more second opcodes specify 64 bit operands.

5. The method according to Claim 4, wherein the arithmetic/logical instruction further comprises the opcode consisting of two separate opcode fields, a first displacement field and a second displacement field, wherein the location in memory is determined by adding contents of the second register to a signed displacement value, the signed displacement value comprising a sign extended value of the first displacement field concatenated to the second displacement field.

6. The method according to Claim 5, further comprising:
responsive to the opcode being a first opcode and the second operand not being on a 32 bit boundary, generating a specification exception; and
responsive to the opcode being a second opcode and the second operand not being on a 64 bit boundary, generating a specification exception.

7. The method according to Claim 6, wherein the processor is a processor in a multi-processor system, further comprising:
the obtaining the second operand comprising preventing other processors of the multi-processor system from accessing the location in memory between said obtaining of the second operand and storing a result at the second location in memory; and
upon said storing the produced result, permitting other processors of the multi-processor system to access the location in memory.

8. A computer program product for executing an arithmetic/logical instruction having an interlocked memory operand, the arithmetic/logical instruction comprising an opcode field, a first register field specifying a first operand in a first register, a second register field specifying a second register, the second register specifying location of a second operand in memory, and a third register field specifying a third register, the computer program product comprising a tangible storage medium readable by a processing circuit and storing instructions which when executed by the processing circuit performs the method as claimed in any of claims 1 to 7.

9. A computer system for executing an arithmetic/logical instruction having an interlocked memory operand, the arithmetic/logical instruction comprising an opcode field, a

first register field specifying a first operand in a first register, a second register field specifying a second register, the second register specifying location of a second operand in memory, and a third register field specifying a third register, comprising:

a memory; and

a processor in communication with the memory, the processor comprising an instruction fetching element for fetching instructions from memory and one or more execution elements for executing fetched instructions, wherein the computer system is configured to perform the method as claimed in any of claims 1 to 7.