

UNITED STATES PATENT AND TRADEMARK OFFICE

BEFORE THE PATENT TRIAL AND APPEAL BOARD

LzLabs GmbH,

Petitioner

v.

INTERNATIONAL BUSINESS MACHINES CORPORATION

Patent Owner

IPR2023-00274
Patent 8,713,289

**DECLARATION OF IAN JESTICE PERTAINING TO PETITION FOR
INTER PARTES REVIEW OF U.S. PATENT NO. 8,713,289**

TABLE OF CONTENTS

I.	INTRODUCTION	1
II.	EXPERIENCE AND QUALIFICATIONS.....	1
III.	TASK SUMMARY AND MATERIALS REVIEWED	5
IV.	LEGAL STANDARDS	5
	A. Level of Ordinary Skill in the Art	5
	B. Legal Standard for Claim Construction	6
	C. Obviousness.....	7
	D. Priority Date	10
V.	OVERVIEW OF THE '289 PATENT	12
	A. Priority Date of the '289 Patent.....	12
	B. Subject Matter of the '289 Patent.....	13
	C. Prosecution History of the '289 Patent	18
VI.	LEVEL OF ORDINARY SKILL IN THE ART.....	21
VII.	OVERVIEW OF THE PRIOR ART	22
	A. Loderer	22
	B. PoO.....	28
	C. Kane.....	36
VIII.	CLAIM CONSTRUCTION	37
IX.	GROUND 1: CLAIMS 1, 4, 6, 10-11, AND 16 ARE OBVIOUS IN VIEW OF LODERER.....	37
	A. Loderer Overview.....	37
	B. Claim 1	44
	1. Claim element 1[a1]: “1. A computer program product for emulating computer instructions from a source machine to produce sequences of instructions on a target machine, said computer program product comprising:”	44

2. Claim element 1[a2]: “a non-transitory computer readable storage medium readable by a processor and storing instructions for execution by the processor for performing a method comprising:” 46
3. Claim element 1[b]: “obtaining by the target machine a computer instruction from the source machine, said source machine having a different architecture from said target machine; and” 47
4. Claim element 1[c]: “generating a sequence of target machine instructions which together operate to derive an encoding for a target machine condition code for the computer instruction, wherein the sequence of target machine instructions provides distinguishing information to distinguish between a plurality of possible outcomes for the target machine condition code, and” 51
5. Claim element 1[d]: “directly calculates the target machine condition code without using branch instructions, the directly calculating comprising:” 56
6. Claim element 1[e]: “determining an intermediate condition code value, the intermediate condition code value being a provisional value for the target machine condition code and subject to change based on the distinguishing information; and”. 58
7. Claim element 1[f]: “determining, based on the intermediate condition code value and based on the distinguishing information, the target machine condition code, wherein at least part of said distinguishing information is separate from the intermediate condition code value.” 60
- C. Claim 4: “The computer program product of claim 1, wherein said method further includes executing said generated sequence of target machine instructions” 62
- D. Claim 6: “The computer program product of claim 1, wherein the sequence of target machine instructions together operate to directly calculate the target machine condition code using at least one of addition, subtraction, and multiplication.” 64

E.	Claim 10: “The computer program product of claim 1, wherein determining the intermediate condition code value comprises storing the intermediate condition code value in a condition code storage location as the provisional target machine condition code, and wherein determining the target machine condition code based on the distinguishing information and the intermediate condition code value comprises performing at least one operation on the intermediate condition code value stored in the condition code storage location to obtain the target machine condition code and storing the target machine condition code in the condition code storage location.”	65
F.	Claims 11 and 16.....	68
X.	GROUND 2: CLAIMS 1-4, 6, 10-12, AND 16-17 ARE OBVIOUS TO LODERER IN VIEW OF POO, KANE, AND THE KNOWLEDGE OF A POSA	77
A.	Motivation to Combine Loderer with PoO and Kane	77
B.	A POSA Would Have Had a Reasonable Expectation of Success Applying Loderer’s Technique to Calculate an Overflow Code for an ADD Instruction Without Branches.....	86
1.	Exemplary ADD Instruction Implementation	86
2.	Detecting the overflow	89
3.	Calculating the overflow condition code	100
C.	Claim 1	102
D.	Claim 2: The computer program product of claim 1, wherein said calculation of the target machine condition code comprises using at least one of carry, sign, and overflow codes in the calculation.	104
E.	Claim 3: The computer program product of claim 2, wherein said calculation employs temporary locations for storing intermediate results.....	108
F.	Claim 4: The computer program product of claim 1, wherein said method further includes executing said generated sequence of target machine instructions.....	109

G.	Claim 6: The computer program product of claim 1, wherein the sequence of target machine instructions together operate to directly calculate the target machine condition code using at least one of addition, subtraction, and multiplication.....	112
H.	Claim 10: The computer program product of claim 1, wherein determining the intermediate condition code value comprises storing the intermediate condition code value in a condition code storage location as the provisional target machine condition code, and wherein determining the target machine condition code based on the distinguishing information and the intermediate condition code value comprises performing at least one operation on the intermediate condition code value stored in the condition code storage location to obtain the target machine condition code and storing the target machine condition code in the condition code storage location.	113
I.	Claims 11, 16.....	114
J.	Claims 12, 17.....	116
XI.	SECONDARY CONSIDERATIONS	116
XII.	CONCLUSION.....	116

LIST OF EXHIBITS

1001	U.S. Patent No. 8,713,289 to Copeland et al (“the ’289 patent”)
1002	Ian Jestice CV
1003	Intentionally Blank
1004	File History of U.S. Patent No. 8,713,289 to Copeland
1005	Translation of EP0843256A2 to Loderer and certification of translation
1006	EP0843256A2 to Loderer
1007	U.S. Patent 6,049,864 to Liu
1008	Enterprise Systems Architecture/390 Principles of Operation (“PoO”), SA22-7201-08, Ninth edition (June 2003)
1009	Kane, G., MIPS RISC Architecture, © 1988 (“Kane”)
1010	U.S. Patent Pub. No. 2006/0036790
1011	U.S. Patent No. 5,889,980
1012	Umeno, H., “New Method for Dispatching Waiting Logical Processors in Virtual Machine System,” IEEE 2005
1013	Scheduling Order (Dkt. 26) <i>International Business Machines Corp. v. LzLabs GmbH and Texas Wormhole, LLC</i> , Case No. 6:22-cv-00299-ADA (W.D. Tex.)
1014	Cragon, H., “Computer Architecture and Implementation,” © 2000, Cambridge University Press, ISBN 0-521-65168-9
1015	Enterprise Systems Architecture/390 Principles of Operation, SA22-7201-06, Seventh Edition (July 1999)
1016	IBM System/370 Principles of Operation, SA22-7085-07, First Edition (March 1983)
1017	Declaration of Jim Mullins, Ph.D

IPR of U.S. Patent 8,713,289

1018	Information Disclosure Statement, U.S. Patent Application No. 09/608750 (June 29, 2005)
1019	Information Disclosure Statement, U.S. Patent Application No. 09/0924755 (July 1, 2005)
1020	U.S. Patent No. 5,661,674
1021	U.S. Patent No. 5,576,555
1022	Plambeck, K. et al., “Development of attributes of z/Architecture,” IBM J. RES. & DEV. VOL. 46 NO. 4/5, (July/September 2002)
1023	U.S. Patent No. 5,481,688
1024	Hennessey, J. et al., “Hardware/Software Tradeoffs for Increased Performance,” ACM SIGARCH Computer Architecture News, Vol. 10, Issue 2, March 1982 (c) 1982
1025	IBM System/360 Principles of Operation, Form A22-6821-0

I. INTRODUCTION

1. My name is Ian Jestice. I make this declaration based upon my own personal knowledge and, if called upon to testify, would testify competently to the matters contained herein.

2. I have been asked to provide technical assistance in the inter partes review of U.S. Patent No. 8,713,289, which I may abbreviate as “the ’289 patent” or refer to as the “Challenged Patent.” This declaration is a statement of my opinions on issues related to the unpatentability of claims of the ’289 patent. I am being compensated at my normal rate of \$575 per hour for my analysis, plus reimbursement for expenses. My compensation does not depend on the content of my opinions or the outcome of this proceeding.

II. EXPERIENCE AND QUALIFICATIONS

3. I hold the equivalent of an undergraduate degree in Telecommunications and Computer Science from the City and Guilds Institute of London, which I obtained in 1971. The City and Guilds Institute of London is an educational organization that was founded in 1878 to develop a national system of technical education in the UK and its commonwealth. It was the required educational organization for engineers working for the government run telecommunications company, the Post Office in the UK. Classes were held at various universities or colleges around the city of London and were taught by

industry professionals or university staff. At the time of my training, the courses included data transmission, assembler programming and traffic analysis.

4. As described in more detail in my *curriculum vitae* (Ex. 1002) I have more than 40 years of industry experience. My areas of expertise include system architecture, software development, diagnostics and testing for IBM 360, 370 and 390 systems, Intel x86 and PowerPC processors and data storage.

5. I have extensive experience verifying compatibility of hardware and software systems against architecture of IBM 360, 370 and 390 mainframe computers. I worked for IBM as a Systems Engineer in the United Kingdom, Canada and California. As part of that work, I debugged mainframe 360 and 370 mainframes at an instruction level. This included detailed analysis of customer failures, down to microcode analysis of logic errors including emulation instruction errors. During this work I frequently had to write machine code level instructions to debug the hardware and microcode. While working at ITEL (later Hitachi Data Systems), I was responsible for assuring the compatibility of ITEL AS/5 to the IBM 370/158 architecture at a machine code level.

6. As part of my experience, I have worked as a design and systems engineer at Amdahl Corporation, IBM and Fujitsu, designing, building, testing and supporting computing environments with storage devices.

7. While working at Amdahl, I was responsible for writing diagnostic

code to assure that Amdahl 5880 and 5890 computer systems correctly executed the IBM 370/390 instruction set. I also personally wrote Amdahl's BUPS (Bring Up Programs) which was used to test the hardware design and gate level simulations. These tests included the instructions to test emulation of earlier architectures such as IBM's 1401 and IBM's 360 systems. I also managed the group that wrote tests to verify memory translation and pipeline management hardware to verify the integrity of the pipeline as a result of different condition codes and branch conditions. While at Amdahl I gave a lecture series on the 370/390 architectures to newly hired design engineers unfamiliar with the systems, particularly in machine check and exception handling.

8. While at Amdahl, I also worked in Japan with Fujitsu, a major investor in Amdahl, verifying their processors and storage devices conformed to the IBM mainframe architecture utilizing a staff group of 15 engineers.

9. I have done considerable work in the area of embedded software systems for industry and consumer products, and other systems including storage products (e.g. Flash Memory (Solid State Disks, memory cards, flash drives), Optical Storage (CD, DVD, WORM, Magneto-Optical), Magnetic Storage (Hard Disk, Floppy Disk, Tape), RAID/Disk Arrays and jukeboxes; USB, SCSI, iSCSI, IDE/ATA/ATAPI/SATA, Fibre Channel), as well as PCMCIA devices, game programming, and software for home appliances and telecommunications devices.

10. I have worked with security in the context of storage devices while working for IBM, Amdahl and Fujitsu. Additionally, while working at Zadian, I was involved in the testing of secure storage devices.

11. In addition to experience in developing and testing interoperable systems that emulated other platforms, I have used emulation systems extensively in development, including in my work as a diagnostic design engineer at Ciena Corporation and Infinera Corporation, where I worked with PowerPC emulators, and in developing code for Android and Apple products, using simulators that target ARM and Apple Processors on an x86 machine.

12. While at the Ciena Corporation and Infinera Corporation I designed, developed and coded the BIOS code, which is the first computer instructions executed after powering on or resetting the system. The BIOS is responsible for testing and initializing the hardware and was written in PowerPC assembler code. I also designed and coded file systems that were responsible for reading and writing data to be saved on persistent storage devices.

13. For these reasons and because of my technical experience and training as outlined in my *curriculum vitae* (Ex. 1002), I am capable of offering technical opinions regarding the design and/or operation of emulation systems, including the support of low-level hardware and system features by such systems.

III. TASK SUMMARY AND MATERIALS REVIEWED

14. I have been asked to review the '289 patent and its prosecution history, to provide an understanding of the technology relevant to the Challenged Patent, to review certain prior-art references, and analyze whether or not those references disclose or teach limitations of claims from the Challenged Patent and whether and why a person of ordinary skill in the art (POSA) would have been motivated to combine or modify such teachings and/or references to arrive at the challenged claims. The opinions stated in this declaration are from the perspective of a POSA.

15. In forming my opinions, I have reviewed the '289 patent, its prosecution history and references cited by the Examiner, the materials cited in the List of Exhibits, and the materials cited throughout my declaration.

IV. LEGAL STANDARDS

A. Level of Ordinary Skill in the Art

16. When interpreting a patent, I understand that it is important to identify the relevant art pertaining to the patent as well as the level of ordinary skill in that art at the time of the claimed invention. The “art” is the field of technology to which the patent is related.

17. I have been instructed by counsel that the person of ordinary skill in the art (“POSA”) is a hypothetical person who is presumed to know the relevant prior art. I understand that the actual inventor’s skill is not determinative of the

level of ordinary skill. I further understand that the factors that may be considered in determining the level of skill include: the types of problems encountered in the art; prior art solutions to those problems; rapidity with which innovations are made; sophistication of the technology; and educational level of active workers in the field. I understand that not all such factors may be present in every case, and one or more of them may predominate.

B. Legal Standard for Claim Construction

18. I understand that the challenged claims of the '289 patent (Ex. 1001) are to be construed in accordance with their ordinary and customary meanings, as would be understood by a person of ordinary skill in the art, in the context of the patent's specification and its file history. For the purposes of this proceeding and the grounds presented herein, Petitioner does not present any constructions and in my opinion, no constructions are needed for the opinions offered herein.

19. I understand that to determine how a person of ordinary skill would understand a claim term, one should look to those sources available that show what a POSA would have understood the disputed claim language to mean. I understand that in construing a claim term, one looks primarily to the intrinsic patent evidence, including the words of the claims themselves, the remainder of the patent specification, and the patent's prosecution history. I understand that extrinsic evidence, which is evidence external to the patent and prosecution history, may

also be useful in interpreting patent claim language when the intrinsic evidence itself is insufficient.

C. Obviousness

20. I understand that the prior art may render a patent claim “obvious.” I understand that two or more prior art references (e.g., prior art articles, patents, or publications) that each disclose fewer than all elements of a patent claim may nevertheless be combined to render a patent claim obvious if the combination of the prior art collectively discloses all elements of the claim and one of ordinary skill in the art at the time would have been motivated to combine the prior art in such a way and would have reasonably expected success in doing so.

21. I understand that this motivation to combine need not be explicit in any of the prior art, but may be inferred from the knowledge of one of ordinary skill in the art at the time the patent was filed. I also understand that one of ordinary skill in the art is not an automaton, but is a person having ordinary creativity. I further understand that one or more prior art references, articles, patents or publications that disclose fewer than all of the elements of a patent claim may render a patent claim obvious if including the missing element would have been obvious to one of skill in the art (e.g., the missing element represents only an insubstantial difference over the prior art or a reconfiguration of a known system).

22. I understand that under the doctrine of obviousness, a claim may be

invalid if the differences between the invention and the prior art are such that the subject matter as a whole would have been obvious at the time the invention was made to a POSA to which the subject matter pertains.

23. To assess obviousness, I understand that I am to consider the scope and content of the prior art, the differences between the prior art and the claim, the level of ordinary skill in the art, and any secondary considerations to the extent they exist.

24. I understand that any evidence of secondary indicia of non-obviousness should be considered when evaluating whether a claimed invention would have been obvious to one of ordinary skill at the time of invention. These secondary indicia of non-obviousness may include, for example:

- a. a long felt but unmet need in the prior art that was satisfied by the claimed invention;
- b. commercial success of processes claimed by the patent;
- c. unexpected results achieved by the invention;
- d. praise of the invention by others skilled in the art;
- e. the taking of licenses under the patent by others; and
- f. deliberate copying of the invention.

25. I understand that there must be a nexus between any such secondary indicia and the claimed invention.

26. It is also my understanding that there are additional considerations that may be used as further guidance as to when the above factors will result in a finding that a claim is obvious, including the following:

- a. the claimed subject matter is simply a combination of prior art elements according to known methods to yield predictable results;
- b. the claimed subject matter is a simple substitution of one known element for another to obtain predictable results; unexpected results achieved by the invention;
- c. the claimed subject matter uses known techniques to improve similar devices or methods in the same way;
- d. the claimed subject matter applies a known technique to a known device or method that is ready for improvement to yield predictable results;
- e. the claimed subject matter would have been “obvious to try” choosing from a finite number of identified, predictable solutions, with a reasonable expectation of success;
- f. there is known work in one field of endeavor that may prompt variations of it for use in either the same field or a different one based on design incentives or other market forces if the variations would have been predictable to a POSA;

- g. there existed at the time of the invention a known problem for which there was an obvious solution encompassed by the patent's claims;
and
- h. there is some teaching, suggestion, or motivation in the prior art that would have led a POSA to modify the prior art reference or to combine prior art reference teachings to arrive at the claimed subject matter.

27. Finally, I understand that a claim may be deemed invalid for obviousness in light of a single prior art reference, without the need to combine references, if the elements of the claim the reference does not expressly disclose are suggested by the reference or are within the knowledge or common sense of one of ordinary skill in the relevant art. In other words, a claim can be obvious in light of a single prior art reference if (i) the reference discloses or suggests each element of the claim and/or (ii) it would have been obvious to supplement or modify the reference based on the knowledge or common sense of a POSA to arrive at the claimed invention.

D. Priority Date

28. I understand that, subject to the next paragraph, the asserted "priority date" of a patent is the earlier of: (a) the date on which a patent application is filed; or (b) the date on which an earlier-filed patent application is filed if the patentee

properly claims the benefit of priority to that earlier-filed patent application.

29. I understand that it is not enough for a patent to merely claim the benefit of an earlier-filed application, but that additional criteria must be met. In particular, the prior application itself must describe the claimed invention, and must do so in sufficient detail that one skilled in the art can clearly conclude that the inventor invented the claimed invention as of the filing date sought. First, I understand that a priority-date analysis is on a claim-by-claim basis. Second, I understand that, for a patent claim to be entitled to the filing date of an earlier patent application, a “Section 112 analysis” must be conducted. I am informed that a “Section 112” analysis encompasses looking to the earlier patent application and ascertaining whether that earlier patent application meets both the written-description and enablement requirements as of the filing date of the earlier application. I understand that it is not enough that the claim would have been obvious from the earlier application, but that application itself must describe the claimed invention.

30. I understand that to satisfy the written description requirement the earlier application must reasonably convey to those skilled in the art that the inventors had possession of the subject matter of the patent as of the filing date of the earlier application. I have been informed that it is the disclosures of the earlier patent application that count, and that while the meaning of terms, phrases, or diagrams in the earlier patent application must be interpreted from the vantage

point of one skilled in the art, all of the claimed limitations must appear in the specification. Further, I understand that this analysis is not a question of whether one skilled in the art might be able to construct the claimed invention from the teachings of the disclosure. The question is not whether a claimed invention is an obvious variant of that which is disclosed in the specification; rather, I understand that an earlier application must itself describe each of the claim limitations.

31. I also understand that to satisfy the enablement requirement, the earlier application must enable a person of ordinary skill in the art to practice the claimed invention without undue experimentation.

V. OVERVIEW OF THE '289 PATENT

A. Priority Date of the '289 Patent

32. I have reviewed the '289 patent, which is entitled "Efficiently emulating computer architecture condition code settings without executing branch instructions." Ex. 1001 (54). I understand the application for the '289 patent was filed on January 30, 2007, and does not claim priority to an earlier filed application. Ex. 1001, (22). I also understand that, in the district court litigation, Patent Owner contends it is entitled to a priority date of March 22, 2006. For purposes of my analysis, I have assumed that the priority date of the '289 patent is January 30, 2007, but my opinions would not change if I instead assumed a priority date of March 22, 2006. I understand that the '289 patent issued on April 29, 2014.

Ex. 1001, (45).

B. Subject Matter of the '289 Patent

33. The '289 patent relates to a system that emulates the generation of condition codes for instructions from a computer having a first architecture on a computer having a different architecture, without using branch instructions. "In virtually all modern data processing systems, the execution of various operations such as arithmetic operations, logical operations and even data transfer operations, may result in the generation of several bits of data to indicate the outcome status of instruction execution. These bits are typically referred to as condition codes." Ex. 1001, 1:18-23.

34. Because they are bits of data that indicate the outcome of instruction execution, condition codes are often used as flags and they can be hardware-specific. Ex. 1001, 1:18-49. They often include such things as overflow, zero result, carry or borrow, and negative result codes. Condition codes may be set as a side effect of the operation many instructions, and the codes often used by subsequent conditional instructions, to take a particular action such as a branch or to move data, based on the condition set. Thus, programmers who work with assembly or machine language programs on machines whose instruction set architectures have condition codes will often make use of the condition codes and conditional branch instructions in the course of routine programming tasks

involving control flow.

35. Many processor instruction set architectures make use of condition codes. Ex. 1024, 2-11; *id.*, 3 (“Many architectures have included condition codes as the primary method of implementing conditional control flow. Condition codes provide an implicit communication between two otherwise disjoint instructions.”). Some processors, such as the MIPS, provide instructions to set registers conditionally, providing an alternate way to set values needed to implement control flow logic. *Id.*, 5.

36. The ’289 patent specification explains that when one processor is emulating another, but does not use the same condition codes, emulating the condition codes “adds considerable space and performance overhead to the emulated instructions.” Ex. 1001, 2:59-61.

37. By way of example, the specification discusses the “conventional handling of condition code generation,” in which the emulator generates a series of if/then statements to determine and set the correct condition code value for a compare instruction. *Id.*, 1:64-2:7. The patent explains that “[i]n conventional approaches to the emulation of a compare instruction, the result is the construction of a sequence of instructions, which include three branch instructions.” *Id.*, 2:7-10. The specification explains that such branch instructions are undesired because branch prediction may make an incorrect guess as to the path the program should

take, and because it may consume branch prediction resources so they are not available for other instruction streams. *Id.*, 2:14-20-32.

38. To avoid using branch instructions, the '289 patent uses sequences of code on the target computer to calculate condition codes that would have been generated on the source computer. "The technique used herein to derive the sequences is to implement very short one or two instruction sequence fragments that set a bit or bits in a result or temporary register to distinguish each possible outcome of a condition code setting. These small code fragments ... are tied together into slightly larger sequences by standard and usually high-performing shifts, rotates and various arithmetic or Boolean instructions available on most computer architectures." *Id.*, 4:22-30. "Very efficient sequences result by avoiding both branch instructions and more complex instructions that are less likely to be optimized in hardware." *Id.*, 4:30-32. The '289 patent suggests using this technique to emulate condition codes of a z/Architecture system on a computer whose processor uses the Intel IA32 architecture or the PowerPC architecture. *Id.*, 3:24-26.

39. As noted in the passages cited above, the use of shifts, rotates, arithmetic instructions, and Boolean instructions is common to most computer architectures. The use of such instructions to manipulate data and put it into a desired form is a routine programming task that commonly arises in computer

programming. Because the specifics as to which such instructions are supported, and the exact behavior of the instructions may vary, programmers often consult references that identify the available instructions and their behavior as part of writing computer code.

40. The '289 patent gives an example of emulating the “Compare Logical” operation that compares the contents of two register operands, rA and rB. Ex. 1001, 5:20-22, 5:29-37. The specification uses examples, written in “assembler pseudocode,” to illustrate behavior. *Id.*, 4:41-48. Pseudocode is often used to illustrate behavior through a made-up language that is easier for a person to read than actual code might be. Because programmers can readily consult references that describe the behavior of the instructions available to them and of their behavior, pseudocode can be a helpful way to convey the idea of program logic, without getting caught up in the details of specific syntax.

41. As seen in Table II below, the emulated condition code in the '289 patent is calculated using a sequence of five instructions:

TABLE II

[1]	sub_set_carry	rC, rA, rB
[2]	set_bit_on_zero	rX, rC
[3]	set_bit_on_not_zero	rC, rC
[4]	add_to_carry_immed	rC, rC, 0
[5]	sub	rC, rC, rX

Instruction	rA > rB Register Contents			rA = rB Register Contents			rA < rB Register Contents		
	rC	rX	Carry	rC	rX	Carry	rC	rX	Carry
[1]	rC > 0	N/A	1	rC = 0	N/A	1	rC < 0	N/A	0
[2]	rC > 0	0	1	rC = 0	1	1	rC < 0	0	0
[3]	1	0	1	0	1	1	1	0	0
[4]	2	0	N/A	1	1	N/A	1	0	N/A
[5]	2	0	N/A	0	1	N/A	1	0	N/A

Id., 5:41-56. Instruction [1] computes a subtraction that sets a carry in register (rC), and instruction [2] sets a register rX=1 if the difference (from instruction [1]) totaled zero and otherwise rX=0, as seen in the yellow highlighting above. *Id.*, 6:22-32. The value in rX is referred to as “distinguishing information” because it is used later (in instruction [5]), to change the value previously stored in rC. *Id.*, 6:37-39, 6:57-58, 7:3-10.

42. Instruction [3] sets a register rC=1 if the difference (from instruction [1]) did not total zero and otherwise rC=0. Ex. 1001, 6:34-36. After instruction [4] (blue), rC is set to “2” if rA>rB, and set to “1” if rA=rB or rA<rB. *Id.*, 6:59-67.

Thus, at this point in the process, the interim condition code contained in rC does not *distinguish* between the cases of rA=rB and rA<rB. Because rX had been set after instruction [2] “to provide discrimination information distinguishing equality from inequality,” the last instruction [5] subtracts rX from rC to distinguish these

two remaining cases (i.e., to distinguish $rA=rB$ and $rA<rB$). *Id.*, 6:37-7:10. As a result of these instructions, rC represents the result (i.e., condition code) of the compare operation: (i) $rC=2$ if $rA>rB$, (ii) $rC=0$ if $rA=rB$, and (iii) $rC=1$ if $rA<rB$. *Id.*

C. Prosecution History of the '289 Patent

43. The application leading to the '289 patent was filed on January 30, 2007. Ex. 1004, 387. The original claims were limited to a target machine that would directly calculate condition codes “from carry, sign and overflow codes” without using branch instructions. *Id.*, 405-408.

44. In a first Office Action, the Examiner rejected claims as anticipated by U.S. Patent 6,049,864 to Liu. Ex. 1004, 226-237 (3/24/09 Office Action, pg. 6). Liu described emulating CISC instructions using a RISC processor, and the Examiner pointed to Liu’s description of the ADDF32 instruction, which generated “carry, overflow, and sign codes.” *Id.*, 233 (3/24/09 Office Action, pg. 7 citing Ex. 1007, Fig. 3). The Examiner found that though Liu did not specifically mention executing at “substantially the same time” as generating the condition codes, as recited in then-pending claims 13, 19, and 25, this was obvious in view of Liu’s disclosure that this occurs at runtime. Ex. 1004, 235-236 (3/24/09 Office Action, pgs. 9-10).

45. The Applicant argued Liu “is totally devoid of any reference

whatsoever to anything that even remotely resembles emulation.” Ex. 1004, 212 (6/16/09 Resp. to Office Action, pg. 12). The Examiner did not find the Applicant’s arguments persuasive and issued a Final Office Action. *Id.*, 178-191 (8/20/09 Final Office Action).

46. The Applicant amended the claims to require that part of the distinguishing information reside outside the intermediate condition code, and then argued:

Liu describes a technique for scheduling a flag generating instruction and a subsequent instruction. In Liu, flags are set by using intermediate flag data produced during execution of an instruction. While Liu describes setting flags, Liu fails to describe, teach or suggest one or more aspects of applicants’ claimed invention. As one example, Liu fails to describe; teach or suggest generating a sequence of target machine instructions which together operate to derive an encoding for a condition code and provide distinguishing information to distinguish a plurality of possible outcomes for the condition code. Further, Liu fails to describe, teach or suggest that at least part of the distinguishing information is separate from the condition code.

Id., 168-169 (12/21/2009 Resp. Office Action, pgs. 4, 11-12). The Examiner responded, observing that Liu disclosed distinguishing information residing outside the condition codes, and maintaining the earlier rejections. *Id.*, 137 (3/14/2012 Office Action, 4 (citing Ex. 1007, 4:53-63)).

47. Next, the Applicant amended the independent claims to recite the

target machine instructions “directly calculates the condition code based, at least in part, on the distinguishing information.” Ex. 1004, 114-120 (6/13/2012 Resp. to Office Action). The Applicant argued “Liu fails to uncover any teaching or suggestion of providing distinguishing information to distinguish between a plurality of possible outcomes for a condition code and directly calculating the condition code, based, at least in part, on this distinguishing information.” *Id.*, 123 (6/13/2012 Resp. to Office Action, pg. 12). The Examiner again rejected the claims in view of Liu, but also indicated then-pending Claims 14, 20, and 26 would be allowable if rewritten in independent form. *Id.*, 95-105 (7/31/2012 Final Office Action). The Examiner recommended that amendments focus on the concept of the condition code and distinguishing information, rather than on minor details of how the instruction is obtained. *Id.*, 104 (7/31/2012 Final Office Action, pg. 9).

48. The Applicant amended the independent claims to recite “determining an intermediate condition code value, the intermediate condition code value being a provisional value for the target machine condition code and subject to change based on the distinguishing information.” Ex. 1004, 74-81 (10/29/2012 Resp. to Office Action, pg. 2). The Applicant stated the specification showed:

a target machine condition code is calculated by, for instance, determining an intermediate condition code value (stored in rC just before execution of the last instruction in the sequence, in the examples presented). The intermediate condition code value is provisionally the

target machine condition code, subject to being changed based on the distinguishing information.

Id., 82 (10/29/2012 Resp. to Office Action, pg. 10, citing paragraphs [0024]-[0032] and Fig. 2 of the application).

TABLE II

[1]	sub_set_carry	rC, rA, rB
[2]	set_bit_on_zero	rX, rC
[3]	set_bit_on_not_zero	rC, rC
[4]	add_to_carry_immed	rC, rC, 0
[5]	sub	rC, rC, rX

Instruction	rA > rB			rA = rB			rA < rB		
	Register Contents			Register Contents			Register Contents		
	rC	rX	Carry	rC	rX	Carry	rC	rX	Carry
[1]	rC > 0	N/A	1	rC = 0	N/A	1	rC < 0	N/A	0
[2]	rC > 0	0	1	rC = 0	1	1	rC < 0	0	0
[3]	1	0	1	0	1	1	1	0	0
[4]	2	0	N/A	1	1	N/A	1	0	N/A
[5]	2	0	N/A	0	1	N/A	1	0	N/A

Ex. 1001, 5:41-56. As shown in Table II (among the portions of the specification cited by the Applicant), the intermediate condition code value is stored in register rC, and the distinguishing information is stored in register rX.

49. The Examiner issued a Notice of Allowance on August 13, 2013 (Ex. 1004, 7), and the '289 patent issued April 29, 2014. Ex. 1001, (45).

VI. LEVEL OF ORDINARY SKILL IN THE ART

50. A POSA at the time the '289 patent was filed (2007) would have had at least the equivalent of undergraduate degree or equivalent in computer science, electrical or computer engineering or related training in computer systems, or

equivalent training from experience working in the field of computer systems. The POSA would also have at least two years of combined engineering, development, or other technical experience related to the development or test of system hardware, software and emulation systems at an assembler or machine code level.

51. A basis for my familiarity with the level of ordinary skill is my own technical experience and my own interaction with large numbers of workers in the field who were at this level of skill. In reaching this opinion as to the hypothetical POSA, I have considered the types of problems encountered in the art, the prior art solutions to those problems, the rapidity with which innovations are made, the sophistication of the technology, and the educational level and professional capabilities of workers in the field. A basis for my familiarity with the level of ordinary skill is also based on my extensive experience hiring those skilled in the field leading up to 2008 and my extensive experience as a consultant since then reinforces my beliefs.

VII. OVERVIEW OF THE PRIOR ART

A. Loderer

52. I understand EP0843256A2 (“Loderer”) published May 20, 1998 and therefore is prior-art. Ex. 1006. Loderer was not considered during prosecution of the '289 patent. I have reviewed and am relying on the translated version of Loderer. Ex. 1005 (translation and certification).

53. Loderer is directed to the same problem as the '289 patent: generating emulated condition codes. Ex. 1005, 1, col. 1 ¶1. Loderer observes that mainframe systems use hardware flags called condition codes that are set by the execution of instructions running on the mainframe architectures, and that when simulating the execution of the mainframe code on processors that do not support the condition codes, the codes must be emulated. *Id.*, col. 1 ¶¶2-3. Loderer solves the problem in essentially the same way as the '289 patent by calculating its encoding of the condition codes on the target computer using a sequence of instructions and without using branch instructions. *Id.*

54. Loderer explains that condition codes can reflect the result of a series of queries or comparisons, and that a typical approach to emulating them would be to use a cascade of “if” statements, resulting in a multi-level branch. Ex. 1005, col. 1 ¶4, col. 5. ¶3. But Loderer explains that these sequences of queries/comparisons followed by branches, which Loderer calls an a “branch cascade,” reduce performance. *Id.*, col. 1 ¶4, col. 2 ¶5. “[W]ith this method, the jump command which for many computers is ‘time consuming’ relative to other machine commands, i.e., requiring a high number of processor clock cycles, results in significant loss of speed.” *Id.*, col. 5 ¶4-col. 6 ¶1. Thus, Loderer concludes that jump commands (branches) are undesirable when emulating the setting of condition codes, because using the branches reduces performance.

55. Loderer describes avoiding branch instructions, using the emulation of the IBM/390 COMPARE instruction on a MIPS processor to illustrate the technique. To avoid branches, Loderer introduces machine instructions that combine a comparison function with the setting of a register (e.g., instructions B1 and B2). Ex. 1005, 1, col. 4 ¶8. “This combination of two sub-functions **in a single machine command** is crucial to avoid branching in the program flow.” Ex. 1005, col. 2 ¶2. A single machine command to perform these sub-functions distinguishes preexisting instructions that may have been implemented in microcode, such as the set equal (SEQ) and set greater than (SGT) instructions. *Id.* The first command results in a register value of 1 for equality and 0 for inequality, with the second command distinguishing $a > 0$ (setting register to 1) from $a < 0$ (setting register to 0). *Id.*, col. 6 ¶¶4-5. Using these instructions, Loderer calculates condition code values without branch instructions.

56. For example, Loderer discloses obtaining a sequence of instructions from a source machine (PCU), whose architecture on hardware M1 differs from that of the target machine (PCZ) architecture on hardware M2.

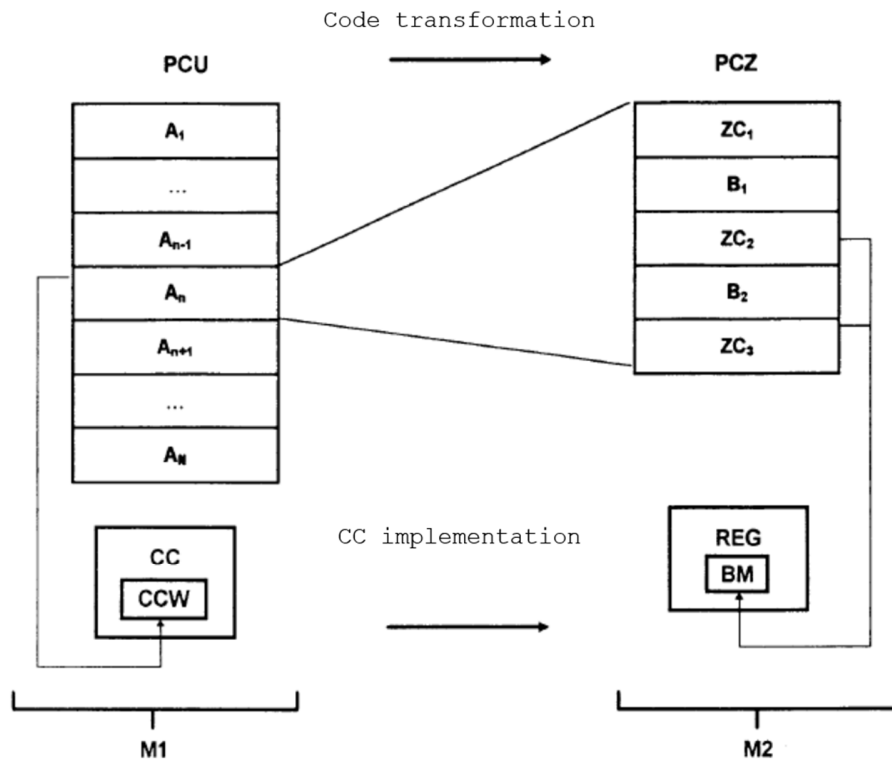


FIG. 1

Ex. 1005, Fig. 1; *Id.*, 1, col. 4 ¶¶7. Instructions $A_1 \dots A_N$ from the source machine with architecture M1 are emulated on the second machine with architecture M2, where each instruction A_n is replaced with a sequence of instructions for architecture M2. *Id.*, col. 4 ¶¶7-8. For example, instruction A_n from the source machine is replaced on the target machine with a “command sequence” consisting “of two commands B1 and B2 required for the semantic representation of the instruction A_n and an additional code consisting of three subsections ZC_1 , ZC_2 and ZC_3 .” *Id.*, col. 4 ¶¶8. Loderer describes an “exemplary embodiment in which an IBM/390 computer system is selected as source hardware,” and emulation is

performed on a “target hardware (e.g. a Mips R4400 RISC machine).” *Id.*, col. 5 ¶¶2-3. In the example illustrated below in Loderer’s Fig. 4, the first subsection, ZC₁, operates to set a register, the second and third subsections ZC₂ and ZC₃ operate to shift the value in a register by a number of bits specified by another register. Because individual instructions that set a value in a register, or that shift the value in a register by a number of bits designated by another register are common machine instructions and are available on the MIPS architecture, a POSA reading the description of the subsections would have considered it likely that each could be implemented using a single machine instruction. *See* §IX.A.

57. Loderer discloses commands (B1 and B2) combine a query/comparison and the setting of a register into a single machine instruction so the emulated code (PCZ) can avoid the need for branch instructions.

Command sequence in PCZ		a = 0	a > 0	a < 0
ZC ₁	Set BM to '0100'	BM = 0100	BM = 0100	BM = 0100
B1	If a = 0, set t ₁ = 1, otherwise t ₁ = 0	t ₁ = 1	t ₁ = 0	t ₁ = 0
ZC ₂	BM << t ₁	BM = 1000	BM = 0100	BM = 0100
B2	If a > 0, set t ₂ = 1, otherwise t ₂ = 0	t ₂ = 0	t ₂ = 1	t ₂ = 0
ZC ₃	BM >> t ₂	BM = 1000	BM = 0010	BM = 0100

FIG. 4

Ex. 1005, Fig. 4; *id.*, col. 6 ¶¶4-5. Loderer shows that instructions B1 and B2 of Fig. 4 above (yellow) combine performing a comparison (e.g., checking whether a=0 or a>0), and setting a register (e.g., setting t₁ or t₂ to either “1” or “0”) into a single machine instruction. “This combination of two sub-functions in a single

machine command is crucial to avoid branching in the program flow.” Ex. 1005, col.2 ¶2. Loderer explains that calculating emulated condition codes without branching improves performance. “The implementation of ... a branch into machine code” is “‘time-consuming’ relative to other machine commands” and “will result in loss of speed.” Ex. 1005, col. 5 ¶4-col. 6 ¶1; *id.*, col. 1 ¶4, col. 1 ¶2, col. 3 ¶2.

58. Loderer’s preferred embodiment applied the technique to the IBM/390 compare instruction on a computer having a MIPS processor, explaining how the machine instructions discussed in Fig. 4 in the target code sequence avoided the cascade of branch instructions. Ex. 1005, col. 6 ¶1, 1. Though the example described above was for the COMPARE instruction, Loderer explicitly states that its technique should be used for “any instruction potentially influencing the value of the condition code.” Ex. 1005, col. 2 ¶1. In addition, Loderer noted that “most of the computer commands influence the value CCW¹ of the condition code CC.” Ex. 1005, col. 4 ¶7. Thus, a POSA reading Loderer would understand that the reference teaches use of the technique for other instructions.

¹ Loderer refers to the “value CCW” of “condition code CC,” which is different from the channel-command word (CCW) used in the PoO.

B. PoO

59. Enterprise Systems Architecture/390 Principles of Operation (“PoO”), SA22-7201-08, Ninth Edition (June 2003) is a publication that describes IBM’s 390 Architecture. Ex. 1008. I understand PoO is prior-art because it was published and publicly accessible more than one year before the priority date of the ’289 patent.

60. For example, PoO’s face indicates public accessibility about June 2003. *See* Ex. 1008, 4 (“Ninth Edition (June, 2003)”; “© Copyright International Business Machines Corporation 1990-2003.”). IBM provided PoO upon request to “your IBM representative or to the IBM branch office serving your locality.” *Id.*, 4, 1025. Listed uses of the publication confirm its public availability: “How did you use this publication?” “an introduction,” “reference manual,” “a text (student),” or “a text (instructor).” *Id.*, 1025. The same was true for previous versions of PoO. Ex. 1022, 13 (“IBM Corporation, z/Architecture Principles of Operation, Order No. SA22-7832; available through IBM branch offices.”); Ex. 1011, 1:18-23 (ESA/390 Principles of Operation “available through IBM branch offices”); Ex. 1020, 1:16-18 (same); Ex. 1021, 1:30-33 (same).

61. IBM also made PoO available through online resources, including IBM’s Online Publications Center. For example, MIPS Technology, Inc. cited PoO in an IDS submitted June 29, 2005, in U.S. Application 09/608,750, and in another

IDS submitted July 1, 2005, in U.S. Application 09/924,755. Exs. 1018, 1019.

Both IDS documents identified “ESA/390 Principles of Operation. IBM Online Publications Center Reference Number SA22-7201-08.” Exs. 1018, 4; Ex. 1019, 4. These documents confirm PoO was accessible electronically at least by July 2005, from at least the IBM Online Publications Center. *Id.*

62. Other publications and patent applications that pre-date the ’289 patent similarly cite PoO, confirming PoO was publicly accessible more than a year before the ’289 patent’s priority date. For example, a 2005 article cited PoO as authority for S/390 instruction behavior and system architecture. Ex. 1012, 5, 8, 10 n.12; Ex. 1017, ¶¶84-105. IBM cited PoO in U.S. Application 10/916,171, filed, August 10, 2004, which IBM incorporated “by reference in its entirety” into the specification. Ex. 1010, [0019]. Earlier versions of PoO were also cited in the prior art, including in a prior art college textbook that provided a student exercise asking students to consult the IBM S/390 documentation to construct a table of instructions. Ex. 1014, 130 (§3.26); Ex. 1017, ¶¶61-83; Ex. 1023, 2:38-41. Such earlier versions of PoO contained similar disclosures (e.g., instructions and condition codes) on which I rely in my analysis below. *Compare* Ex. 1008, 941-942 (C-1, C-2) *with* Ex. 1015, 855-856 (C-1, C-2) and Ex. 1016, 519 (C-1).

63. During my career, I recall IBM making versions of the Principles of Operations available to customers, developers, and others. While working with

customers of IBM, ITTEL and Amdahl as a customer engineer and as a systems engineer, my customers programmers and I would have access to the IBM Principles of Operation for both assembler programming and debug. Software and sometimes even hardware errors frequently resulted in what is known as a “dump.” This was a printout of the volatile storage or RAM containing the machine code and the data in hexadecimal format. One of the tasks for programmers, systems engineers and customer engineers was to analyze the dump to troubleshoot problems. As the dump was in machine code, the only complete reference document to understand hexadecimal code that I was aware of, was the Principles of Operation. During this time, we used IBM/360, IBM/370, IBM/390 architecture Principles of Operations. It is my understanding that IBM routinely made the Principles of Operation available to customers, developers, and others interested in obtaining them, including via IBM’s online websites or upon request to IBM. In my opinion, a POSA would have known that such IBM documents were the most detailed and authoritative source of technical information for IBM systems, including instructions and their condition codes, including POSAs interested in emulating programs written for IBM mainframes.

64. PoO is a machine reference manual that describes each instruction and function of the IBM/390 architecture. *See* Ex. 1008. Beginning with the System 360, IBM has published incremental versions of PoO, which includes sufficient

detail concerning the instructions of IBM's architecture and of their operating behavior to support low-level programming. *See* Ex. 1025.

65. In the IBM/390 instruction set, the condition code value may indicate a numeric overflow for certain functions (e.g., arithmetic functions). For example, Fig. C-1, shows the condition code values for the ADD instruction which uses condition code of "3" to indicate a numeric overflow.

Instruction	Condition Code			
	0	1	2	3
ADD (gen)	Zero	< zero	> zero	Overflow
ADD (BFP)	Zero	< zero	> zero	NaN
ADD DECIMAL	Zero	< zero	> zero	Overflow
ADD HALFWORD	Zero	< zero	> zero	Overflow
ADD HALFWORD IMMEDIATE	Zero	< zero	> zero	Overflow
ADD LOGICAL	Zero, no carry	Not zero, no carry	Zero, carry	Not zero, carry
ADD LOGICAL WITH CARRY	Zero, no carry	Not zero, no carry	Zero, carry	Not zero, carry
ADD NORMALIZED	Zero	< zero	> zero	--
ADD UNNORMALIZED	Zero	< zero	> zero	--
AND	Zero	Not zero	--	--
CANCEL SUBCHANNEL	Function initiated	--	--	Not operational
CHECKSUM	Checksum complete	--	--	CPU-determined completion
CIPHER MESSAGE	Normal completion	--	--	Partial completion
CIPHER MESSAGE WITH CHAINING	Normal completion	--	--	Partial completion
CLEAR SUBCHANNEL	Function initiated	--	--	Not operational
COMPARE (gen, HFP)	Equal	Low	High	--
COMPARE (BFP)	Equal	Low	High	Unordered
COMPARE AND FORM CODEWORD	Equal	OCB=0: low OCB=1: high	OCB=0: high OCB=1: low	--
COMPARE AND SIGNAL	Equal	Low	High	Unordered
COMPARE AND SWAP	Equal	Not equal	--	--

Figure C-1 (Part 1 of 6). Summary of Condition-Code Settings

Ex. 1008, 941. For the ADD instruction, which performs a signed addition of two operands to produce a sum, condition code values 0-2 indicate that the sum is: (a) equal to zero (condition code=0), (b) less than zero (condition code=1), or (c) greater

than zero (condition code=2). *Id.*; *See also id.*, 316 (7-12). The condition code for the ADD and COMPARE instructions are similar as highlighted above. The difference is that an ADD instruction may give rise to an overflow, in which case the corresponding condition code is 3. *Id.*, 941; *see id.*, 316 (describing signed ADD operation and overflow conditions). By contrast, the COMPARE instruction does not give rise to an overflow and thus does not use condition code 3 (see “--” in table above for condition code 3 column for the COMPARE instruction). *Id.*, 941-942.

66. The above-cited portions of PoO are content that has long been present in earlier versions of IBM’s published Principles of Operation. For example, in the Seventh Edition (dated on its face July 1999), Appendix C of SA22-7201-06 shows the same condition codes for the ADD, Compare, and other instructions.

Appendix C. Condition-Code Settings

This appendix lists the condition-code setting for instructions in ESA/390 which set the condition code. In addition to those instructions listed which set the condition code, the condition code is set unpredictably by PROGRAM RETURN, and it may be changed by DIAGNOSE and the target of EXECUTE. The condition code is loaded by LOAD PSW, by SET PROGRAM MASK, and by an interruption. The condition code is set to zero by initial CPU reset and is loaded by the successful conclusion of the initial-program-loading sequence.

The condition codes for the vector facility are not included in this appendix. See the publication *IBM Enterprise Systems Architecture/390 Vector Operations*, SA22-7207 for the condition codes set by vector instructions.

Some models may offer instructions which set the condition code and do not appear in this document, such as those provided for assists or as part of special or custom features.

Instruction	Condition Code			
	0	1	2	3
ADD (gen), ADD HALFWORD, ADD HALFWORD IMMEDIATE	Zero	< zero	> zero	Overflow
ADD (BFP)	Zero	< zero	> zero	NaN
ADD DECIMAL	Zero	< zero	> zero	Overflow
ADD LOGICAL	Zero, no carry	Not zero, no carry	Zero, carry	Not zero, carry
ADD NORMALIZED	Zero	< zero	> zero	--
ADD UNNORMALIZED	Zero	< zero	> zero	--
AND CHECKSUM	Zero	Not zero	--	--
CLEAR SUBCHANNEL	Checksum complete	--	--	CPU-determined completion
COMPARE (gen, HFP)	Function initiated	--	--	Not operational
COMPARE (BFP)	Equal	Low	High	Unordered
COMPARE AND SIGNAL	Equal	Low	High	Unordered
COMPARE HALFWORD, COMPARE HALFWORD IMMEDIATE	Equal	Low	High	--
COMPARE AND FORM CODEWORD	Equal	OCB=0: low OCB=1: high	OCB=0: high OCB=1: low	--
COMPARE AND SWAP	Equal	Not equal	--	--
COMPARE DECIMAL	Equal	Low	High	--
COMPARE DOUBLE AND SWAP	Equal	Not equal	--	--
COMPARE LOGICAL	Equal	Low	High	--
COMPARE LOGICAL CHARACTERS UNDER MASK	Equal	Low	High	--
COMPARE LOGICAL LONG	Equal	Low	High	--

Figure C-1 (Part 1 of 4). Summary of Condition-Code Settings

© Copyright IBM Corp. 1990, 1991, 1993, 1994, 1996, 1997, 1998, 1999

C-1

Ex. 1015, 855.

67. Though the typesetting places the content off by a few lines, the description of the ADD instruction of both documents is the same. Ex. 1008, 316 (7-12), Ex. 1015, 296 (7-10). Likewise, the description of the COMPARE instruction matches. Ex. 1008, 339 (7-35); Ex. 1015, 309 (7-23).

68. Looking to earlier versions of PoO, Page C-1 of Appendix C of SA22-7085-0, the Principles of Operation document for the IBM IBM/370 that preceded the IBM/390 generation of machines, shows substantially the same table with substantially the same content.

APPENDIX C. CONDITION-CODE SETTINGS

This appendix lists the condition-code setting for all instructions in the 370-XA architecture which set the condition code. In addition to those instructions listed which set the condition code, the condition code may be changed by DIAGNOSE and the target of

EXECUTE. The condition code is loaded by LOAD PSW, by SET PROGRAM MASK, and by an interruption. The condition code is set to zero by initial CPU reset and is loaded by the successful conclusion of the initial-program-loading sequence.

Instruction	Condition Code			
	0	1	2	3
ADD, ADD HALFWORD ADD DECIMAL ADD LOGICAL ADD NORMALIZED ADD UNNORMALIZED	zero zero zero, no carry zero zero	< zero < zero not zero, no carry < zero < zero	> zero > zero zero, carry > zero > zero	overflow overflow not zero, carry -- --
AND CLEAR SUBCHANNEL COMPARE (gen, fl pt) COMPARE HALFWORD COMPARE AND SHAP	zero function accepted equal equal equal	not zero -- low low not equal	-- -- high high --	-- unsuccessful -- -- --
COMPARE DECIMAL COMPARE DOUBLE AND SWAP COMPARE LOGICAL COMPARE LOGICAL CHARACTERS UNDER MASK COMPARE LOGICAL LONG	equal equal equal equal equal equal	low not equal low low low	high -- high high high	-- -- -- -- --
EDIT, EDIT AND MARK EXCLUSIVE OR HALT SUBCHANNEL INSERT ADDRESS SPACE CONTROL INSERT CHARACTERS UNDER MASK	zero zero function accepted zero all zeros	< zero not zero status pending, function not executed one first bit one	> zero -- busy -- first bit zero	-- -- unsuccessful -- --
LOAD ADDRESS SPACE PARAMETERS LOAD AND TEST (gen, fl pt) LOAD COMPLEMENT (gen) LOAD COMPLEMENT (fl pt) LOAD NEGATIVE (gen, fl pt)	parameters loaded zero zero zero zero	primary ASN not available < zero < zero < zero < zero	secondary ASN not available or not authorized > zero > zero > zero --	space-switch event -- overflow -- --
LOAD POSITIVE (gen) LOAD POSITIVE (fl pt) LOAD REAL ADDRESS MODIFY SUBCHANNEL MOVE LONG	zero zero translation available function executed length equal	-- -- ST entry invalid status pending, function not executed length low	> zero > zero PT entry invalid busy length high	overflow -- length violation unsuccessful destr overlap

Summary of Condition-Code Settings (Part 1 of 2)

Appendix C. Condition-Code Settings C-1

Ex. 1016, 519 (C-1). In this 1983 version of the document, the description of the ADD instruction (*id.*, 180 (7-8) and of the COMPARE instruction (*id.*, 186 (7-14)) is the same as in the S/390 versions.

69. Even versions of PoO dating back to the IBM/360, which published in the 1960s, disclose this information. Ex. 1025 shows portions of the condition

code table for fixed-point arithmetic and logical operations (e.g., ADD, COMPARE), and that they were the same as described in the later documents cited above. Ex. 1025, 24, 51. Similarly, the description of the ADD instruction (Ex. 1025, 27), and of the COMPARE instruction (*id.*, 29) are substantially the same.

70. It is not surprising that the same disclosure has appeared in dozens of versions of these documents over the years. In order to make programs backward compatible as new versions of its mainframe systems are released, IBM has maintained consistency in many of its features.

71. Thus, for decades, persons of ordinary skill in the art who wanted to know what instructions were supported by a given version of IBM's 360/370/390 or Z architecture machines, would have known to look to the Principles of Operation for more than fifty years. In this report, I have specifically cited Enterprise Systems Architecture/390 Principles of Operations, SA22-7201-08, Ninth edition (June 2003), as it is a publicly available version of the document that was available and dating from a few years before the '289 patent. Ex. 1008. In my opinion, it would have been obvious to a POSA to rely on the same publicly available content, taken from any of the myriad published versions of the Principles of Operations, as a POSA would likely have been aware that the operating behavior and condition codes of the ADD and COMPARE instructions, as addressed in my report, were substantially the same.

C. Kane

72. “MIPS RISC Architecture” by Gerry Kane (“Kane”) is a book that describes the MIPS architecture, including the behavior of MIPS instructions and techniques for writing machine code for the MIPS computer. Ex. 1009. I understand that Kane was published in 1989 and is thus prior-art. *Id.*, 3; Ex. 1017, ¶¶34-60.

73. Kane includes a section on testing whether an arithmetic operation resulted in an overflow. Ex. 1009, 45 (C-8, “Testing for Overflow”). Kane discloses that signed addition and subtraction trap if an overflow occurs. *Id.* To detect overflow without traps, Kane discloses specific MIPS instruction sequences for detecting overflow for various arithmetic functions. *Id.*, 45 (C-8), 56 (C-9). A two-step test detects overflow for signed addition. “During addition, overflow occurs if the signs of the addends are the same and the sign of the sum is different.” *Id.*, 45 (C-8) (emphasis added). Kane explains that MIPS handles overflows using software (a system trap) because the MIPS architecture “provides no condition code register containing status bits such as ... Overflow.” *Id.*, 43 (C-1), 45 (C-8). “[I]f it is necessary to detect signed overflow without using traps,” Kane gives example code for overflow detection for signed addition:

```

Signed Addition

/* compute t0 = t1 + t2, branch to L on signed overflow */
addu  t0, t1, t2  /* compute sum */
xor   t3, t1, t2  /* if operands have different signs */
bltz  t3, if      /* then overflow not possible */
/* t1 and t2 have same sign */
xor   t3, t0, t1  /* if sum does not also have the same sign */
bltz  t3, L       /* then addition overflowed */
/* nop */
L:

```

Id., 45 (C-8), 46 (C-9).

VIII. CLAIM CONSTRUCTION

74. In my opinion, and for purposes of this declaration, the claim terms of the '289 patent do not require construction to address the issues presented. In my analyses of the '289 patent, I have applied the ordinary meaning of the claim language as it would have been understood to a POSA in 2007 in view of the patent specification and its prosecution history.

IX. GROUND 1: CLAIMS 1, 4, 6, 10-11, AND 16 ARE OBVIOUS IN VIEW OF LODERER

A. Loderer Overview

75. Like the '289 patent, Loderer describes an emulation system that generates emulated condition codes. Ex. 1005, 1. As explained in §VII.A above, Loderer illustrates emulation of condition codes for IBM/390 instructions on a computer having a MIPS processor by generating sequences of instructions on the target machine that calculate the codes without branch instructions.

76. Loderer uses the setting of condition codes for IBM/390 comparison

instructions to illustrate the problem and solution. Fig. 2 illustrates condition code behavior for the COMPARE instruction:

Comparison result	CCW	BM
OP1 equal to OP2	0	1000
OP1 < OP2	1	0100
OP1 > OP2	2	0010
Overflow	3	0001

FIG. 2

Ex. 1005, Fig. 2. The S390 comparison operation compares two signed operands (OP1 and OP2) and sets a condition code value (column CCW in Fig. 2 above) to zero, one, or two, depending on whether OP1 is equal to, less than, or greater than OP2. *Id.*, col. 5 ¶¶1-3, Fig. 2. Loderer encodes the condition codes in the values of register BM on the target machine, with BM=1000 for CCW=0, BM=0100 for CCW=1, and BM=0010 for CCW=2. *Id.* Although “an overflow does not matter” for the COMPARE instruction, the overflow is encoded as BM=0001 for CCW=3. *Id.*, col. 5 ¶3. A POSA would recognize that there could be different ways to encode the condition code, such as a two-bit binary representation, a 4-bit bit map, or others. Additional bits could be used in an encoding scheme, for example, if one desired to include error detection or error correction in the encoding. The use of so-called redundant bits is routinely done to encode data that must be accurately recalled from storage or transmitted between interconnected devices in computer systems.

Loderer's use of a 4-bit encoding, with a one stored in one location and zeros stored in the other locations is an encoding of the CCW. It would have been obvious to a POSA to encode the CCW in this manner, to allow the bits of the BM register to directly indicate whether or not a particular condition had occurred, without the need to read and decode the value from a binary representation.

77. Loderer first describes an “obvious and commonly used” method for generating condition code BM for comparing first operand (a) to the numerical value “0”:

<u>a equal to 0 ?</u>		
yes	no	
Set BM to '1000'	<u>a > 0 ?</u>	
	yes	no
	Set BM to '0010'	Set BM to '0100'

FIG. 3

Ex. 1005, Fig. 3, col. 1 ¶¶1, 4; col. 5 ¶¶1-2. The method uses a branching cascade (e.g., a series of “if-then” statements), to set the correct value of BM. *Id.*, col. 1 ¶4. To begin, one checks, as seen in the top cell of the table in Fig. 3, whether $a=0$, and if so, sets $BM=1000$, as indicated by the cell on the lower left. *Id.*, col. 5 ¶4, Fig. 3. If $a \neq 0$, “a jump command, with which the second individual query is called” to check if $a > 0$, and if so, set $BM=0010$, else set $BM=0100$. *Id.* In other words, if $a \neq 0$, then the code would jump over the instruction that set $BM=1000$, to an instruction that checks whether $a > 0$. If $a > 0$, then, in the lower-middle cell of Fig. 3, the register BM

is set so that BM=0010. If a is not greater than zero, then another jump (branch) would go to an instruction that set BM=0100. Loderer refers to this approach as a branching cascade because of the sequence of branch instructions, which Loderer explains will tend to slow down execution.

78. To avoid the loss of performance due to the branch instructions, Loderer's Fig. 4 illustrates a target machine (PCZ) command sequence that calculates target condition codes without branch instructions. Ex. 1005, Fig. 4, col. 5 ¶4-col. 6 ¶2. "FIG. 4 depicts the command sequence generated in the program code PCZ for the target hardware M2 and consisting of the subsections ZC₁, ZC₂, ZC₃ of the additional code and the commands B1 and B2." *Id.*, col. 6 ¶2.

Command sequence in PCZ		a = 0	a > 0	a < 0
ZC ₁	Set BM to '0100'	BM = 0100	BM = 0100	BM = 0100
B1	If a = 0, set t ₁ = 1, otherwise t ₁ = 0	t ₁ = 1	t ₁ = 0	t ₁ = 0
ZC ₂	BM << t ₁	BM = 1000	BM = 0100	BM = 0100
B2	If a > 0, set t ₂ = 1, otherwise t ₂ = 0	t ₂ = 0	t ₂ = 1	t ₂ = 0
ZC ₃	BM >> t ₂	BM = 1000	BM = 0010	BM = 0100

FIG. 4

Id., Fig. 4. Commands B1 and B2 (yellow) combine performing a comparison (e.g., a=0 or a>0) with setting a register (e.g., t₁ or t₂) into a single machine instruction. Loderer explains "[t]his combination of two sub-functions in a single machine command is crucial to avoid branching in the program flow." *Id.*, col. 2 ¶2. Thus, by using the machine instructions that Loderer introduced to combine the comparison

and the setting of a register value in a single instruction, Loderer avoids the need to branch, as in the branch cascade described with respect to Fig. 3.

79. Loderer steps through the command sequence, explaining how intermediate values of the target condition code value are set by commands ZC_1 and ZC_2 , with the final value set by command ZC_3 . *Id.*, col. 6 ¶¶2-5.

80. The start of the command sequence, ZC_1 , sets BM to an intermediate value of 0100. Ex. 1005, col. 6 ¶3. This intermediate value of the condition code is subject to change based on the values of intermediate information (the values of t_1 and t_2) set by the commands that follow. Command B1 sets t_1 , the first bit of distinguishing information, equal to 1 if $a=0$, and equal to zero if a is not equal to zero:

Command B1 represents a first command needed to simulate the comparison instruction present in the program code PCU and serves to check the equality of both operands ($OP1 = a$, $OP2 = 0$). The result of this first individual query is stored as the result value “1” (for equality) or “0” (for inequality) in the register variable t_1 .

Ex. 1005, col. 6 ¶4. This instruction avoids the branch instruction Loderer discussed would appear in an “obvious” implementation with respect to Fig. 3. *Id.* It is distinguishing information because, as explained below, the value of t_1 is used by the next instruction to distinguish between two different cases.

81. Next, ZC_2 shifts BM to the left by the value stored in register t_1 .

Because $t_1=0$ unless $a=0$ (the result of command ZC_1), BM is only shifted if $a=0$, thereby distinguishing the case of $a=0$. *Id.*, col. 6 ¶4. If $t_1=0$, ZC_2 does not shift BM. *Id.* The shift left instruction will shift zeros in from the right, when the bits of the register are shifted to the left.

82. Next, command B2 sets t_2 , a second bit of distinguishing information, equal to 1 if $a>0$, and equal to zero if $a<0$:

The subsequent command B2 represents the second command needed to simulate the comparison instruction present in the program code PCU. It checks whether the first operand ($OP1 = a$) is greater than the second one ($OP2 = 0$). The result of this second individual query is stored as result value "1" (for $a > 0$) or "0" (for $a < 0$) in the register variable t_2 .

Ex. 1005, col. 6 ¶5. As with command B1, a branch instruction is avoided. ZC_3 uses t_2 , as seen in the fifth row of Fig. 4, to shift BM to the right if $t_2=1$, thereby distinguishing the case of $a>0$ from the case of $a<0$. *Id.*

83. The value stored in the BM register (initially 0100) is subject to change based on the information stored in registers t_1 and t_2 . *Id.*, Fig. 4, 6 ¶¶4-5. Depending on the values of t_1 and t_2 , the intermediate condition code stored in BM may be shifted left or right to generate the target machine condition code. *Id.*

84. As explained in §VII.A, “commands” B1 and B2, and “subsections” ZC_1 , ZC_2 , and ZC_3 are listed in the “Command sequence in PCZ” for the target

machine PCZ shown in Fig. 4. Ex. 1005, Fig. 4. B1 and B2 are single machine instructions, as explained above. *Id.*, col. 2 ¶2; *Id.*, col. 6 ¶¶4-5. Although ZC₁, ZC₂, and ZC₃ are referred to as “subsections” they are also listed in the “Command sequence in PCZ.” Ex. 1005, Fig. 4, col. 6 ¶¶3-5, col. 4 ¶8. ZC₁ corresponds with setting the value in register BM, and ZC₂ and ZC₃ correspond with the first and second parts of a “two-part bit operation” implemented with a shift operation on t₁ and t₂. *Id.*, col. 6 ¶¶4-5, Fig. 4 (ZC₁ corresponding with “Set BM to ‘0100’”; ZC₂ corresponding with “BM<<t₁”; and ZC₃ corresponding with “BM>>t₂”). In my opinion, a POSA would understand that ZC₁, ZC₂, and ZC₃ would be implemented using specific instructions of the target architecture (the architecture of the target hardware M2, e.g., MIPS machine). For example, setting bits in a register (the function of ZC₁) is a routine processor operation performed by machine instructions, and can be accomplished by individual MIPS instructions—e.g., Load Byte (LB), Load Byte Unsigned (LBU), Load Halfword (LH), Load Halfword Unsigned (LHU), Load Upper Immediate (LUI), Load Word (LW), Add Immediate (ADDI), Add Immediate Unsigned (ADDUI), Or Immediate (ORI), and Exclusive Or Immediate (XORI). Ex. 1009, 22 (A10), 23 (A-11), 27-32 (A35-A40), 35 (A-58), 42 (A-86). If the desired bit pattern was previously stored in a register, a move instruction could also be used to achieve this result. Shifting bits in a register left or right (the function of ZC₂ and ZC₃) is another routine processor

operation performed by machine instructions, and can be accomplished with individual MIPS instructions—e.g., Shift Left Logical (SLL), Shift Left Logical Variable (SLLV), Shift Right Logical (SRL), and Shift Right Logical Variable (SRLV). *Id.*, 36 (A-62), 37 (A-63), 39 (A-70), 40 (A-71). A POSA would have been motivated to use one of the existing MIPS instructions to implement ZC₁, ZC₂, and ZC₃ because the instructions implement the corresponding functions described in Loderer with a single instruction and do not require additional programing and/or testing. Moreover, based on the explicit teaching in Loderer to avoid branches, a POSA would understand that any implementation of ZC₁, ZC₂, and ZC₃ would be without branches.

B. Claim 1

1. Claim element 1[a1]: “1. A computer program product for emulating computer instructions from a source machine to produce sequences of instructions on a target machine, said computer program product comprising:”

85. As explained in §VII.A and §IX.A, Loderer discloses a system that emulates instructions from a source machine, to produce sequences of instructions on a target machine. “The invention relates to a method for simulating a condition code when porting hardware-specific program code from a source hardware to a target hardware, in which no hardware support of a condition code is provided.” Ex 1005, col. 1 ¶1.

86. “Code is converted into program code PCZ for a target hardware M2.”

Ex. 1005, col. 4 ¶8.

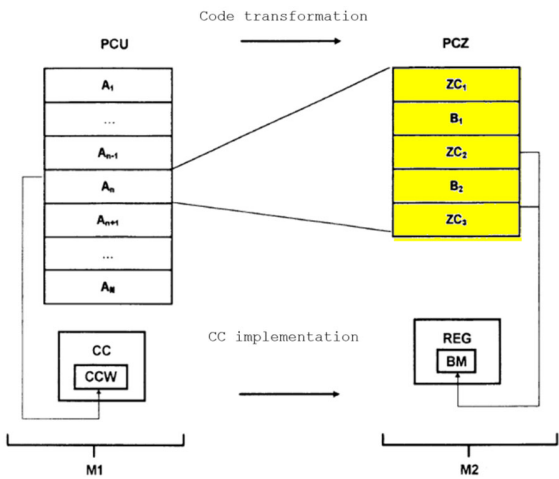


FIG. 1

Id., Fig. 1.

Command sequence in PCZ		a = 0	a > 0	a < 0
ZC ₁	Set BM to '0100'	BM = 0100	BM = 0100	BM = 0100
B1	If a = 0, set t ₁ = 1, otherwise t ₁ = 0	t ₁ = 1	t ₁ = 0	t ₁ = 0
ZC ₂	BM << t ₁	BM = 1000	BM = 0100	BM = 0100
B2	If a > 0, set t ₂ = 1, otherwise t ₂ = 0	t ₂ = 0	t ₂ = 1	t ₂ = 0
ZC ₃	BM >> t ₂	BM = 1000	BM = 0010	BM = 0100

FIG. 4

Id., Fig. 4.

87. The “PCZ” command sequences highlighted in yellow in Figs. 1 and 4 above illustrate sequences of instructions executing on the target machine (M2). Ex 1005, 1, Figs. 1, 4. A POSA would understand that code running on a computer means that the instructions reside in a memory, thus disclosing and suggesting a computer program product. Moreover, a POSA would have understood, in view of Loderer’s characterization of the sequence as having been made for the target

machine (M2) to emulate the execution of instructions from the source machine (M1) that Loderer is referring to a sequence of program instructions running on the target machine (M2), and that sequences of instructions running on a processor are stored in a computer memory.

88. To the extent the preamble is limiting, a POSA would understand Loderer discloses and suggests it in the emulator running on a target computer (M2) executing target machine instructions (e.g., the instructions shown in Figs. 1 and 4). Ex. 1005, 1, col. 2 ¶1, Figs. 1, 4. And, for the reasons given above, a POSA would also understand the preamble is also obvious in view of Loderer.

2. Claim element 1[a2]: “a non-transitory computer readable storage medium readable by a processor and storing instructions for execution by the processor for performing a method comprising:”

89. As explained in §VII.A and the paragraphs directly above in §IX.B.1, Loderer discloses an emulator that runs program instructions on the processor of a target machine. “When porting the program code of the source hardware, any instruction potentially influencing the value of the condition code is transformed by being simulated in the program code for the target hardware by means of a command generating a result value in the course of the execution of the program and without using branching commands.” Ex. 1005, col. 2 ¶1. A POSA would understand this disclosure of program instructions running on the target machine discloses and suggests the program is stored in a non-transitory computer readable

storage medium readable by the processor of the target machine.

90. Loderer discloses that the emulator, running on a MIPS processor, may simulate the generation of condition codes for emulation of IBM/390 hardware. Ex. 1005, col. 1 ¶2, col. 5 ¶3. A POSA would understand that a simulator emulating a complex instruction set computer (CISC) processor, such as one that implements the IBM/390 architecture, on a reduced instruction set computer (RISC) processor such as a MIPS processor, uses software read from non-transitory memory. A POSA would further understand that a MIPS processor does not inherently contain the instructions described in Loderer for emulating the condition codes, and that it would be obvious that such software instructions would be stored in a non-transitory memory before they are loaded onto the processor of system M2.

91. Accordingly, Loderer discloses or renders obvious element 1[a2].

3. Claim element 1[b]: “obtaining by the target machine a computer instruction from the source machine, said source machine having a different architecture from said target machine; and”

92. Loderer discloses a source machine having a first architecture (designated M1) from which the instructions originate, and a target machine having different architecture (designated M2) that is simulating execution of the source program. Ex. 1005, 1, col. 4 ¶¶7-8, claim 1. A POSA would recognize that M1 and M2 as described in Fig. 1 below have different architectures because M2

does not support the condition codes of M1, and because Loderer specifically describes embodiments in which M1 is the IBM/390 architecture and M2 is the MIPS architecture. *Id.*, col. 1 ¶¶2-3, col. 5 ¶3, claim 1. “The invention relates to a method for simulating a condition code when porting hardware-specific program code from a source hardware to a target hardware, in which no hardware support of a condition code is provided.” *Id.*, col. 1 ¶1.

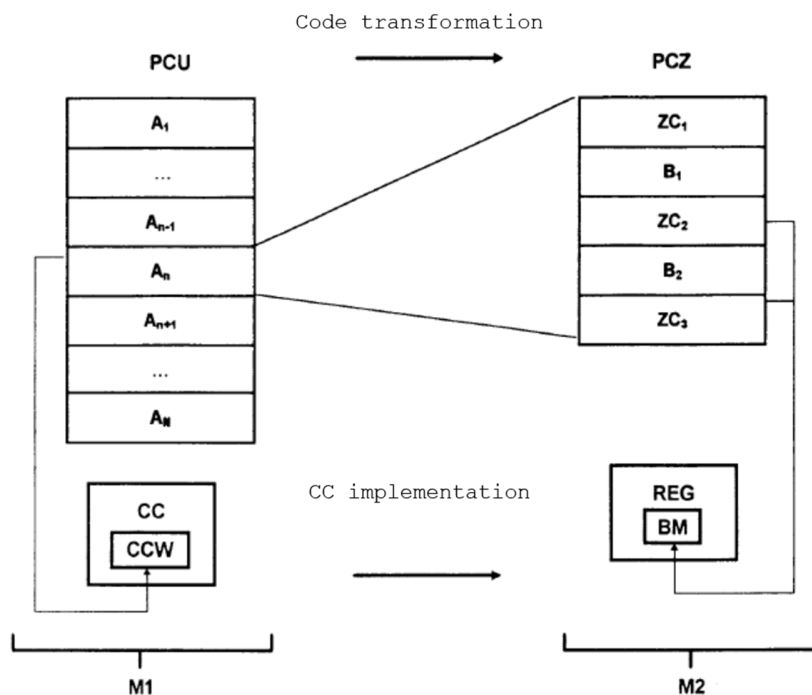


FIG. 1

Id., Fig. 1. Instructions $A_1...A_N$ from the machine having architecture M1 are emulated on the second machine having architecture M2, where each instruction A_n is replaced with a sequence of instructions. *Id.*, col. 4 ¶7. For example, instruction A_N is replaced with a “command sequence” consisting of two commands B_1 and B_2 :

In a code transformation with which the program code PCU to be ported is converted into program code PCZ for a target hardware M2, each of the instructions $A_1, A_2, \dots, A_N$ is implemented as a command (or a semantically equivalent command sequence) in the program code PCZ. For instructions potentially influencing the value CCW of the condition code CC, additional program code (additional code) is generated for the simulation of the condition code CC on the target hardware M2. In the example shown in FIG. 1, in this way, the instruction A_n in the program code PCU is implemented as a command sequence in the program code PCZ. This command sequence consists of two commands B_1 and B_2 required for the semantic representation of the instruction A_n and an additional code consisting of three subsections ZC_1, ZC_2 and ZC_3 .

Id., col. 4 ¶8.

93. The translation takes place on the target machine:

The simulation of the condition code on the target hardware according to the current invention is carried out as follows: First, mapping is defined which uniquely associates each possible value of the condition code on the source hardware with a bit pattern on the target hardware...When porting the program code of the source hardware, any instruction potentially influencing the value of the condition code is transformed by being simulated in the program code for the target hardware by means of a command generating a result value in the course of the execution of the program and without using branch commands.

Id., col. 2 ¶1. Because Loderer discloses that these steps occur on the target machine, it is obvious to a POSA that the target machine obtains the instructions (e.g. instructions $A_1 \dots A_N$) from the source machine (M1) in order to translate them and to emulate the appropriate condition codes.

94. Loderer explains that the IBM/390 computer system is an exemplary source machine, and the MIPS R4400 RISC machine is a target machine. Ex. 1005, col. 5 ¶¶1-2 col. 1 ¶¶2-3. Loderer states that RISC architecture (i.e., the target machine) “usually have no condition code” so “when transforming hardware-specific program code from a source hardware with condition code to program code for such a target hardware without condition code, a software level conditional code must be simulated.” *Id.*, col. 1 ¶3.

95. The translation steps disclosed in Loderer occur on the target machine (*id.*, col. 2 ¶¶1-2), confirming the target machine obtains the source machine instructions because the target machine (e.g., MIPS RISC machine) is simulating the execution of the IBM/390 instructions of the source machine. *See id.*, col. 2 ¶1 (“any instruction potentially influencing the value of the condition code is transformed by being simulated in the program code for the target hardware by means of a command generating a result value in the course of the execution of the program.”), *id.*, col. 7 claim 1 (“each instruction (A_n) influencing the value (CCW) of the condition code (CC) in the program code (PCU) for the source hardware

(M1) is simulated in the program code (PCZ) for the target hardware (M2)”).; col.

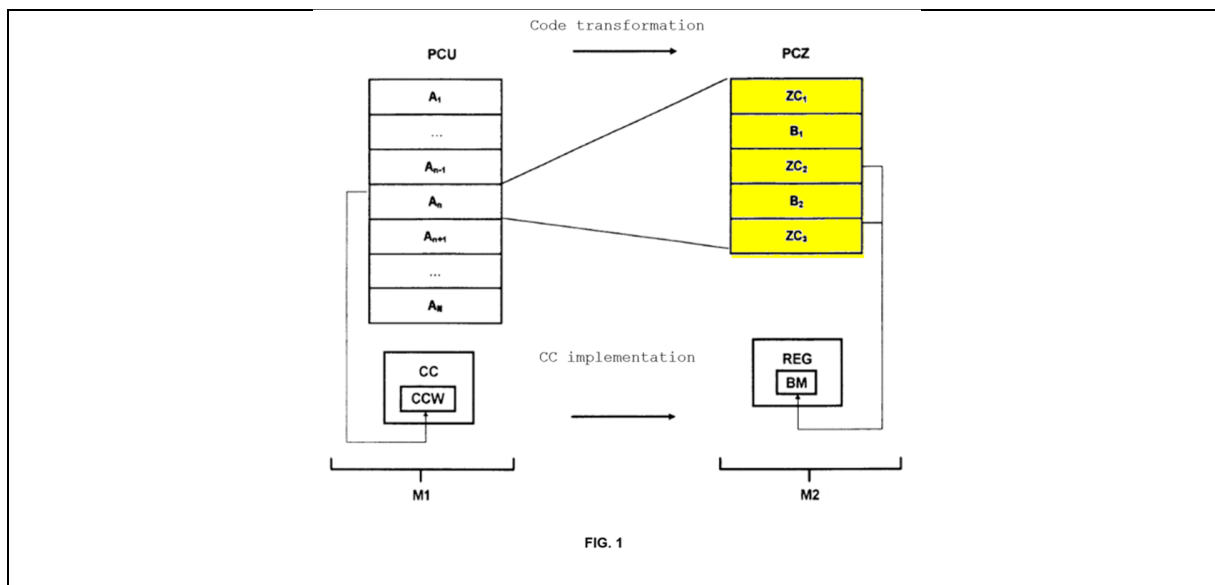
2 ¶5, col. 3 ¶4, col. 4 ¶8.

96. Accordingly, Loderer discloses 1[b].

4. Claim element 1[c]: “generating a sequence of target machine instructions which together operate to derive an encoding for a target machine condition code for the computer instruction, wherein the sequence of target machine instructions provides distinguishing information to distinguish between a plurality of possible outcomes for the target machine condition code, and”

97. As explained in §IX.A and 1[b], Loderer discloses generating a sequence of target machine instructions that together operate to derive an encoding for a target machine condition code.

98. Loderer’s Fig. 1 depicts a sequence of target machine instructions (yellow) that derive an encoding for a target machine condition code in register BM for the source instruction A_n .



Ex. 1005, Fig. 1, col. 4 ¶¶7-8

99. As explained in §VII.A and §IX.A, the same illustrative sequence of target instructions is depicted below in Fig. 4 (yellow). Commands B1 and B2 provide distinguishing information and stored it in registers t_1 and t_2 (blue), for use by commands ZC_2 and ZC_3 . Ex. 1005, col. 6 ¶¶2-5. The distinguishing information distinguishes between the different values of the condition code, as explained in §VII.A above and in §IX.B.7 below.

	Command sequence in PCZ	$a = 0$	$a > 0$	$a < 0$
ZC_1	Set BM to '0100'	BM = 0100	BM = 0100	BM = 0100
B1	If $a = 0$, set $t_1 = 1$, otherwise $t_1 = 0$	$t_1 = 1$	$t_1 = 0$	$t_1 = 0$
ZC_2	BM $\ll t_1$	BM = 1000	BM = 0100	BM = 0100
B2	If $a > 0$, set $t_2 = 1$, otherwise $t_2 = 0$	$t_2 = 0$	$t_2 = 1$	$t_2 = 0$
ZC_3	BM $\gg t_2$	BM = 1000	BM = 0010	BM = 0100

FIG. 4

Ex. 1005, Fig. 4.

100. For example, the distinguishing information stored in t_1 by the B1 instruction distinguishes between the possible outcome of $a=0$, and the possible outcomes of $a \neq 0$. Likewise, the distinguishing information stored in t_2 by the B2 instruction in the instruction sequence distinguishes between the possible outcome of $a > 0$ and $a < 0$. *Id.*, col. 6 ¶4.

101. The sequence of target machine instructions illustrated in Figs. 1 and 4 (labeled “Command sequence in PCZ” above) operate together to derive a

representation for the target machine condition code that is stored in register BM.

Ex. 1005, Fig. 4, col. 4 ¶6.

102. The values of the BM register are a target encoding of the condition codes because they are a machine representation of the source condition codes that directly map to the corresponding values on the IBM/390 computer (i.e., the source machine). *Id.*, 1, col. 1 ¶2. “Mapping is defined which uniquely associates each possible value (CCW) of the condition code (CC) on the source hardware (M1) with a bit pattern (BM) on the target hardware (M2). To store the bit pattern (BM) associated with the respective current value (CCW), a memory area (REG) is provided on the target hardware (M2).” *Id.*, 1.

Comparison result	CCW	BM
OP1 equal to OP2	0	1000
OP1 < OP2	1	0100
OP1 > OP2	2	0010
Overflow	3	0001

FIG. 2

Id., Fig. 2. “[T]he condition code simulated by the additional code is stored in the form of an associated bit pattern BM.” *Id.*, col. 5 ¶1. Generating a form of the condition code that can be accessed from the register in which it is stored is generating an encoding of the target condition code.

103. Fig. 2 illustrates the mapping of emulated condition code word

(CCW) values to the corresponding register BM values (the encoding for the target machine). *Id.*, col. 5, ¶3. “[E]ach value *i* of the original condition code is associated, on the target hardware, with a bit pattern in which only a bit position uniquely associated with the value *i* is set to ‘1’ and all other bits are set of the bit pattern to zero. This special association of bit patterns is particularly advantageous when these bit patterns correspond to the jump masks of the program branching instructions on the source hardware (e.g., in case of IBM/390 systems).” *Id.*, col. 3 ¶4. See also the discussion of encoding above at §VII.A.

104. The ’289 patent uses “distinguishing information” to refer to data used to determine whether an “intermediate condition code value” should be changed to generate the “target machine condition code value.” Support for this understanding of “distinguishing information” can be found in the language of the claims, and in the examples of the specification. For example, claim 1 recites “the sequence of target machine instructions provides **distinguishing information to distinguish between a plurality of possible outcomes for the target machine condition code.**” Ex. 1001, 9:31-34. As explained in §V.B, Table II of the ’289 patent discloses an example distinguishing information of a bit stored in a register rX that may be used to alter the contents of the register rC containing an intermediate value of the target machine condition code stored in a register rX. *Id.*, 6:22-39, 6:57-58, 7:3-10. During prosecution, the Applicant described target machine

condition code will be held in register rC, and the intermediate condition code value stored in rC is “subject to change based on the distinguishing information (the value stored in register rX in these examples).” Ex. 1004, 83 (10/29/2012 Res. to Office Action).

105. Loderer likewise uses distinguishing information to distinguish between a plurality of possible outcomes for the target machine condition code. The start of the command sequence, ZC₁, sets BM to an intermediate value of 0100:

	Command sequence in PCZ	a = 0	a > 0	a < 0
ZC ₁	Set BM to '0100'	BM = 0100	BM = 0100	BM = 0100
B1	If a = 0, set t ₁ = 1, otherwise t ₁ = 0	t ₁ = 1	t ₁ = 0	t ₁ = 0
ZC ₂	BM << t ₁	BM = 1000	BM = 0100	BM = 0100
B2	If a > 0, set t ₂ = 1, otherwise t ₂ = 0	t ₂ = 0	t ₂ = 1	t ₂ = 0
ZC ₃	BM >> t ₂	BM = 1000	BM = 0010	BM = 0100

FIG. 4

Ex. 1005, Fig. 4, col. 6 ¶3. This intermediate value corresponds to the value “1” of the condition code, and is subject to change based on the values of intermediate information and based on the distinguishing information (the values of t₁ and t₂) set by the commands that follow. *Id.*, col. 6 ¶¶3-5. Command B1 sets t₁, the first bit of distinguishing information, equal to 1 if a=0, and equal to 0 if a is not equal to zero. *Id.*, col. 6 ¶4. Command ZC₂ shifts BM to the left by the value in t₁, thereby distinguishing the case of a=0 (i.e., t₁=1) which shifts BM to the left by 1 from the

case of $a \neq 0$ (i.e., $t_1=0$) which does not shift BM. *Id.* Next, command B2 sets t_2 , a second bit of distinguishing information, equal to 1 if $a > 0$, and equal to zero if $a < 0$. *Id.*, col. 6 ¶5. Command ZC₃ shifts BM to right by the value in t_2 , if $t_2=1$, thereby distinguishing the case of $a > 0$. *Id.*

106. Loderer thus discloses distinguishing information (t_1 and t_2) to distinguish between a plurality of outcomes (e.g., $a=0$, $a > 1$, and $a < 1$).

107. The examples of the '289 patent operate in similar manner to Loderer, generating a sequence of five instructions that construct the target condition code in a register (rC), and may modify the condition code based on data stored in another register (rX), to distinguish different cases (e.g., $rC=2$ if $rA > rB$, $rC=0$ if $rA=rB$, and $rC=1$ if $rA < rB$), Ex. 1001, 5:38-7:10; §V.B.

108. Accordingly, Loderer discloses 1[c].

5. Claim element 1[d]: “directly calculates the target machine condition code without using branch instructions, the directly calculating comprising:”

109. In the '289 patent, the target machine condition codes are calculated by sequences of instructions that include, for example, getting or setting bits, bit manipulations, Boolean operations, arithmetic operations, and moves. Ex. 1001, 4:46-7:49.

110. As explained in §VII.A, §IX.A, and in 1[c] above, Loderer similarly discloses using command sequences that directly calculate the target machine

condition code **without** branch instructions. *See* Ex. 1005, 1; col. 2 ¶¶1, 5, col. 3, ¶1 col. 6 ¶2, Fig. 4. Loderer explicitly states “any instruction potentially influencing the value of the condition code is transformed by being simulated in the program code for the target hardware by means of a command generating a result value in the course of the execution of the program **and without using branching commands.**” Ex. 1005, col. 2 ¶1.

For instructions potentially influencing the value CCW of the condition code CC, additional program code (additional code) is generated for the simulation of the condition code CC on the target hardware M2. In the example shown in FIG. 1, in this way, **the instruction A_n in the program code PCU is implemented as a command sequence in the program code PCZ.** This command sequence consists of two commands B1 and B2 required for the semantic representation of the instruction A_n and an additional code consisting of three subsections ZC₁, ZC₂ and ZC₃.

Id., col. 4 ¶8.

111. Fig 4 shows instruction ZC₁ sets an initial intermediate setting of BM=0100, and that, depending on the values of the distinguishing information generated by commands B1 and B2 and stored in registers t₁ and t₂ (blue), the shift instructions ZC₃ and ZC₄ directly calculate the values of the condition codes without branches. Ex. 1005, col. 6 ¶¶2-5.

	Command sequence in PCZ	a = 0	a > 0	a < 0
ZC ₁	Set BM to '0100'	BM = 0100	BM = 0100	BM = 0100
B1	If a = 0, set t ₁ = 1, otherwise t ₁ = 0	t ₁ = 1	t ₁ = 0	t ₁ = 0
ZC ₂	BM << t ₁	BM = 1000	BM = 0100	BM = 0100
B2	If a > 0, set t ₂ = 1, otherwise t ₂ = 0	t ₂ = 0	t ₂ = 1	t ₂ = 0
ZC ₃	BM >> t ₂	BM = 1000	BM = 0010	BM = 0100

FIG. 4

Id., Fig. 4.

112. Thus, Loderer discloses the execution of the command sequence on the target machine, as depicted in Fig. 4, that directly calculates the target machine condition code without using branch instructions. Element 1[d] recites the “the directly calculating” comprises the determining steps of 1[e] and 1[f], which as shown below, are disclosed in Loderer and also performed through same direct calculation. Accordingly, Loderer discloses element 1[d].

6. Claim element 1[e]: “determining an intermediate condition code value, the intermediate condition code value being a provisional value for the target machine condition code and subject to change based on the distinguishing information; and”.

113. As explained in 1[b]-1[c], and as depicted in Fig. 4 below (yellow), Loderer discloses determining intermediate condition code values which are provisional values for the target machine condition code.

Command sequence in PCZ		a = 0	a > 0	a < 0
ZC ₁	Set BM to '0100'	BM = 0100	BM = 0100	BM = 0100
B1	If a = 0, set t ₁ = 1, otherwise t ₁ = 0	t ₁ = 1	t ₁ = 0	t ₁ = 0
ZC ₂	BM << t ₁	BM = 1000	BM = 0100	BM = 0100
B2	If a > 0, set t ₂ = 1, otherwise t ₂ = 0	t ₂ = 0	t ₂ = 1	t ₂ = 0
ZC ₃	BM >> t ₂	BM = 1000	BM = 0010	BM = 0100
FIG. 4				

Ex. 1005, Fig. 4.

114. The first intermediate condition code value, BM=0100, is set by ZC₁, the first command in Fig. 4. *Id.*, col. 6 ¶3. This provisional value in the BM register is subject to change in later instructions (e.g., ZC₂ and ZC₃) based on the distinguishing information stored in t₁ and t₂.

115. For example, ZC₂ may change the first provisional value in register BM to another value by shifting BM based on the value of t₁ (distinguishing information). Ex. 1005, col. 6 ¶4. The value of t₁ distinguishes between the case of a=0 and the cases in which a is not equal to zero. If t₁=1, ZC₂ shifts the bits in BM to the left by one, thereby distinguishing the a=0 case. *Id.*

116. Similarly, the value of BM calculated by instruction ZC₂ in the third row of Fig. 4 is also a provisional value for the target machine condition code as it is subject to change based on the distinguishing information of t₂. Ex. 1005, col. 6 ¶4. For example, command ZC₃ shifts BM to the right if t₂=1. *Id.*, col. 6 ¶5.

117. Accordingly, Loderer discloses determining an intermediate condition

code value (e.g., value in register BM), the intermediate condition code value being a provisional value for the target machine condition code and subject to change based on the distinguishing information (e.g., the values in t_1 and t_2).

7. Claim element 1[f]: “determining, based on the intermediate condition code value and based on the distinguishing information, the target machine condition code, wherein at least part of said distinguishing information is separate from the intermediate condition code value.”

118. As explained in §V.B above, the '289 patent describes the data stored in register rX as distinguishing information and illustrates by example why it is distinguishing information. Ex. 1001, 6:37-58. Register rC, on the other hand, holds the intermediate condition code value.

TABLE II

[1]	sub_set_carry		rC, rA, rB	
[2]	set_bit_on_zero		rX, rC	
[3]	set_bit_on_not_zero		rC, rC	
[4]	add_to_carry_immed		rC, rC, 0	
[5]	sub		rC, rC, rX	

Instruction	rA > rB			rA = rB			rA < rB		
	Register Contents			Register Contents			Register Contents		
	rC	rX	Carry	rC	rX	Carry	rC	rX	Carry
[1]	rC > 0	N/A	1	rC = 0	N/A	1	rC < 0	N/A	0
[2]	rC > 0	0	1	rC = 0	1	1	rC < 0	0	0
[3]	1	0	1	0	1	1	1	0	0
[4]	2	0	N/A	1	1	N/A	1	0	N/A
[5]	2	0	N/A	0	1	N/A	1	0	N/A

Id., 5:24-27, 5:38-56. Though rX was copied from rC (the register that ultimately stores the condition code) in instruction [2], the specification notes the setting of rX “occurs outside of (that is apart from) both rC and the CPU carry bit.” *Id.*, 6:57-58.

The value stored in rX distinguishes between conditions: e.g., rX=1 when rA=rB, otherwise rX=0. Thus, data stored in another register (i.e., a register distinct from the register that stores the condition code) that distinguishes between cases of the condition codes is an example of distinguishing information that is separate from the intermediate condition code value. *See* §VII.A, §IX.A.

119. As explained above in 1[c] and 1[d] above, Loderer discloses calculating a target machine condition code (e.g., BM) based on the intermediate condition code value (e.g., intermediate value stored in BM), and the distinguishing information (e.g., values stored in t₁ and t₂). The two registers (t₁ and t₂) that store the distinguishing information are different than the register (BM) that stores the intermediate condition code value. Ex. 1005, col. 6 ¶¶2-5.

120. Loderer states that the condition code is stored in its own “separate register” that “is provided on the target hardware.” *Id.*, col. 2 ¶2, Fig. 1. The distinguishing information (t₁ and t₂), is stored in “two register variables.” *Id.*, col. 6 ¶2, Fig. 4. In my opinion, a POSA would understand the use of different register names or labels (e.g., t₁, t₂, BM) reflects that the information (e.g., the distinguishing information in registers t₁ and t₂) is separate from the intermediate condition code value (e.g., in register BM). Thus, the distinguishing information stored in registers t₁ and t₂ is separate from the intermediate condition code value in BM, similar to register rC and rX disclosed in the '289 patent.

121. Accordingly, Loderer discloses or suggests element 1[h].

122. Claim 1 is obvious because Loderer, in view of the knowledge of a POSA, discloses or suggests every element of the claim.

C. Claim 4: “The computer program product of claim 1, wherein said method further includes executing said generated sequence of target machine instructions”

123. For the reasons described in §IX.A and 1[c]-1[f], Loderer discloses generating a sequence of target machine instructions to emulate condition codes on the target machine and executing these instructions to compute the required values.

124. For example, Loderer Figures 1 and 4 disclose execution of the sequence of target machine instructions (program code PCZ) for execution on target hardware M2.

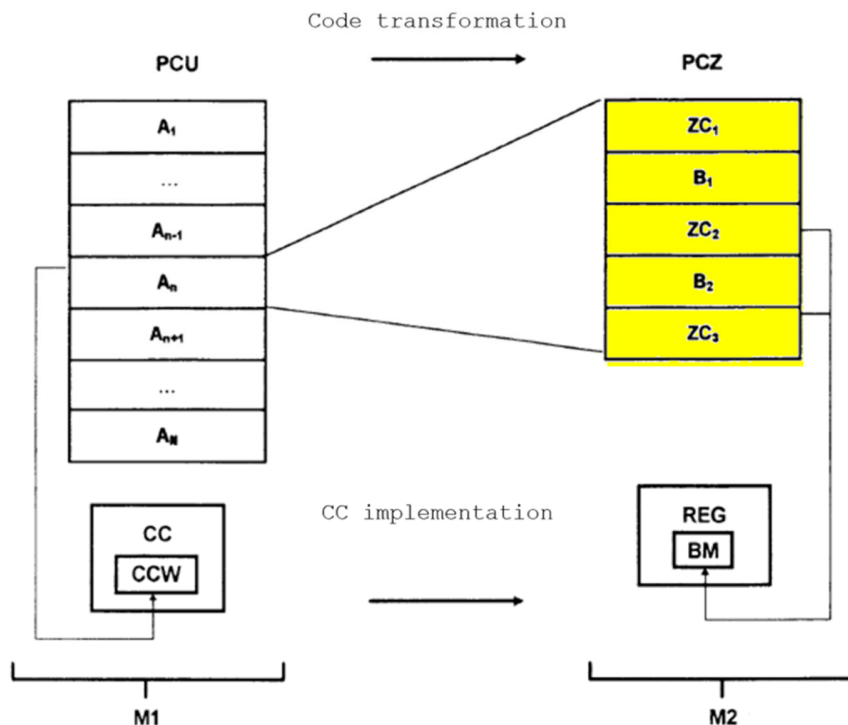


FIG. 1

Ex. 1005, Fig. 1; *see also id.*, Fig. 4. In the emulation of the COMPARE instruction, it is the execution of the instruction sequence that generates the intermediate condition codes, the distinguishing information, and ultimately generates the condition code value. *Id.*, col. 6 ¶¶2-5, Fig. 4.

125. Loderer not only explains that the execution of the sequence of instructions generates the condition code value in the case of the COMPARE example, but also that the execution of a sequence of instructions that operate without branching should be done to generate the condition code value for other instructions. “[A]ny instruction potentially influencing the value of the condition code is transformed by being simulated in the program code for the target hardware

by means of a command generating a result value **in the course of the execution of the program** and without using branching commands.” *Id.*, col. 2 ¶1; *see also id.*, col. 7 claim 1 (“each instruction (A_n) influencing the value (CCW) of the condition code (CC) in the program code (PCU) for the source hardware (M1) is simulated in the program code (PCZ) for the target hardware (M2) without using branching commands and using at least one command (B1, B2) **which generates a result value during the program execution**”), 1 (“generating a result value in the course of the execution of the program”), col. 2 ¶4 (“During the transformation, additional code, hereinafter referred to as ‘additional code’, is generated for the target hardware, by which at runtime the bit pattern corresponding to the value of the condition code on the source hardware after execution of the instruction is ascertained.”), col. 1 ¶3.

126. Thus, Loderer discloses the elements of claim 4.

D. Claim 6: “The computer program product of claim 1, wherein the sequence of target machine instructions together operate to directly calculate the target machine condition code using at least one of addition, subtraction, and multiplication.”

127. As explained in 1[d] above, Loderer discloses a sequence of target machine instructions that directly calculate target condition codes for the compare instruction using a shift instruction. Ex. 1005, 1, col. 2 ¶¶1, 4, col. 7 claim 1, col. 1 ¶3. Shifting a binary number one place to the left multiplies the number by two, directly calculating the next interim value of the condition code. The ’289 patent

includes a pseudocode instruction that it calls double: “[3] double rC”. Ex 1001, 7:33. The specification describes the operation of this “double” instruction in Table I, and a POSA would recognize that a left shift instruction would be a readily available example of a computer instruction that is commonly available in computer instructions sets to perform the double operation described in the pseudocode. *Id.*, 4:61 (instruction “double rX, rY” “doubles the value in rY and places the result in rX”). For at least these reasons, a POSA would understand a shift left instruction, set to shift one place to the left would be an example of using a multiplication to directly calculate the condition code.

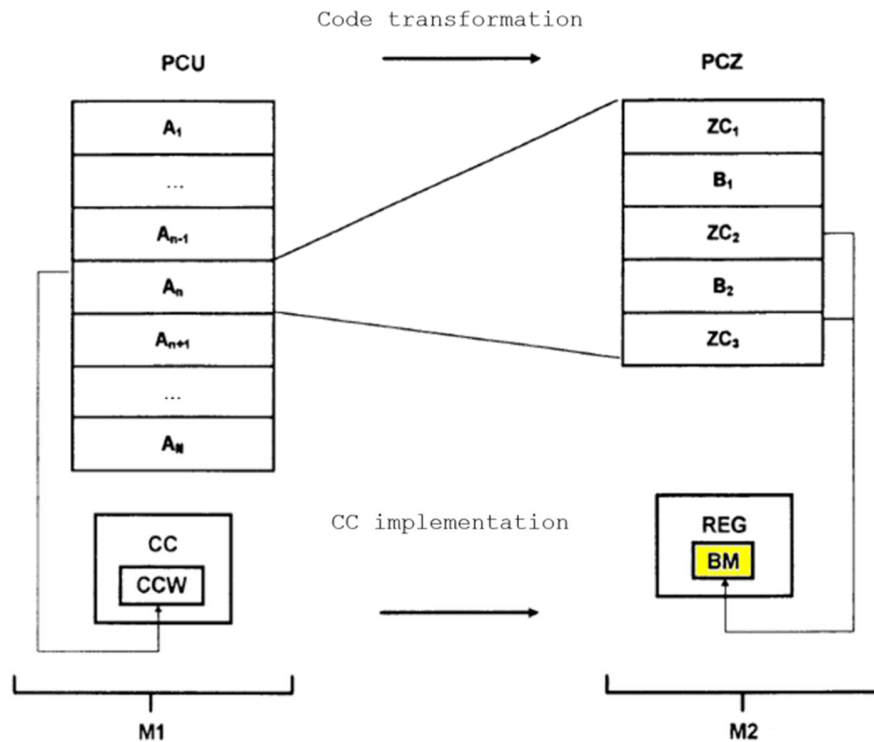
128. Loderer similarly uses a shift left instruction in the same manner. For example, when a=0, Loderer takes the preliminary value of BM=0100, which is a decimal value 4, and shifts it one place to the left (BM=1000) in command ZC₂, which is a decimal value of 8. Ex. 1005, col. 2 ¶¶2-5, Fig. 4. Thus, shifting BM by one bit to the left is equivalent to multiplying BM by two. *Id.* Thus, Loderer discloses using multiplication (by a binary shift) in the sequence of target machine instructions that calculate the target machine code.

129. Accordingly, Loderer discloses the element of claim 6.

E. Claim 10: “The computer program product of claim 1, wherein determining the intermediate condition code value comprises storing the intermediate condition code value in a condition code storage location as the provisional target machine condition code, and wherein determining the target machine condition code based on the distinguishing information and the intermediate condition code value

comprises performing at least one operation on the intermediate condition code value stored in the condition code storage location to obtain the target machine condition code and storing the target machine condition code in the condition code storage location.”

130. As explained in 1[e], Loderer discloses storing an intermediate condition code value in register BM, a condition code storage location.



Ex. 1005, Fig. 1; *see also id.*, 1, col. 5 ¶1, col. 2 ¶2.

131. The intermediate condition code value stored in BM is a provisional target machine condition code because it is subject to change as the sequence of machine instructions executes. *Id.*, col. 2 ¶¶2-5, Fig. 4. The BM register is the condition code storage location because, as it stores the computed encoding of the condition code, as explained in 1[c] and 1[e]-[f] above. *Id.*

132. Loderer discloses several provisional values of the target machine condition code are stored in BM. Ex. 1005, Fig. 4, col. 6 ¶¶3-5. Distinguishing information stored in t_1 and t_2 is applied to each these intermediate condition code values, respectively, to calculate the final target machine condition code. The first such intermediate condition code value of BM=0100 is stored in the BM register by ZC₁. *Id.* A second intermediate condition code value is generated by ZC₂, and the final value of the target condition code is set by command ZC₃. *Id.* As discussed for 1[c], commands ZC₂ and ZC₃ use distinguishing information stored in t_1 and t_2 respectively, to selectively perform shift operations on the intermediate code value stored in the condition code storage location BM. *Id.*

133. Distinguishing information (t_1 and t_2) is used to distinguish between outcomes in which the first and second operands are equal, the first operand is greater than the second, or the first operand is less than the second operand. *Id.*, col. 6 ¶¶4-5, Fig. 4. In the case in which $a=0$, such that $t_1=1$ and $t_2=0$, the intermediate code stored in BM is shifted to the left, producing the value BM=1000. On the other hand, in the case in which the first operand a is greater than the second operand (0), $t_1=0$ and $t_2=1$. *Id.* This case results in command ZC₃ shifting the intermediate code to the right, producing the value BM=0010. *Id.* Finally, in the case in which the first operand is less than the second operand (0), the values of $t_1=t_2=0$. *Id.* In this case, instructions ZC₂ and ZC₃ shift by zero,

leaving the intermediate code unaltered. *Id.*

134. Thus, Loderer discloses storing the intermediate condition code value in a condition code storage location (e.g., the intermediate value stored in register BM) as the provisional target machine condition code, and performing at least one operation on the intermediate condition code value stored in the condition code storage location (e.g., commands ZC₂ and ZC₃ that shift the value in BM using distinguishing information t₁ and t₂) to obtain the target machine condition code and storing the target machine condition code in the condition code storage location (e.g., the target machine condition code is stored in register BM, the same register that held the intermediate condition code).

135. Accordingly, Loderer discloses the elements of claim 10.

F. Claims 11 and 16

136. Independent claims 11 and 16 contain similar limitations as claim 1:

1[a1]. A computer program product for emulating computer instructions from a source machine to produce sequences of instructions on a target machine, said computer program product comprising:	11. A computer system for emulating computer instructions from a source machine to produce sequences of instructions on a target machine, the computer system comprising:
---	--

1[a2]. a non-transitory computer readable storage medium readable by a processor and storing instructions for execution by the processor for performing a method comprising:	a memory; and a processor in communication with the memory, wherein the computer system is capable of performing a method, the method comprising:
1[b]. obtaining by the target machine a computer instruction from the source machine, said source machine having a different architecture from said target machine; and	obtaining by the target machine a computer instruction from the source machine, said source machine having a different architecture from said target machine; and
1[c]. generating a sequence of target machine instructions which together operate to derive an encoding for a target machine condition code for the computer instruction, wherein the sequence of target machine instructions provides distinguishing information to distinguish between a plurality of possible outcomes for the target machine condition code, and	generating a sequence of target machine instructions which together operate to derive an encoding for a target machine condition code for the computer instruction, wherein the sequence of target machine instructions provides distinguishing information to distinguish between a plurality of possible outcomes for the target machine condition code, and

1[d]. directly calculates the target machine condition code without using branch instructions, the directly calculating comprising:	directly calculates the target machine condition code without using branch instructions, the directly calculating comprising:
1[e]. determining an intermediate condition code value, the intermediate condition code value being a provisional value for the target machine condition code and subject to change based on the distinguishing information; and	determining an intermediate condition code value, the intermediate condition code value being a provisional value for the target machine condition code and subject to change based on the distinguishing information; and
1[f]. determining, based on the intermediate condition code value and based on the distinguishing information, the target machine condition code, wherein at least part of said distinguishing information is separate from the intermediate condition code value.	determining, based on the intermediate condition code value and based on the distinguishing information, the target machine condition code, wherein at least part of said distinguishing information is separate from the intermediate condition code value.

1[a1]. A computer program product for emulating computer instructions from a source machine to produce sequences of instructions on a target machine, said computer program product comprising:	16. A method for emulating computer instructions from a source machine to produce sequences of instructions on a target machine, said method comprising:
1[a2]. a non-transitory computer readable storage medium readable by a processor and storing instructions for execution by the processor for performing a method comprising:	
1[b]. obtaining by the target machine a computer instruction from the source machine, said source machine having a different architecture from said target machine; and	obtaining by the target machine a computer instruction from the source machine, said source machine having a different architecture from said target machine; and

1[c]. generating a sequence of target machine instructions which together operate to derive an encoding for a target machine condition code for the computer instruction, wherein the sequence of target machine instructions provides distinguishing information to distinguish between a plurality of possible outcomes for the target machine condition code, and	generating a sequence of target machine instructions which together operate to derive an encoding for a target machine condition code for the computer instruction, wherein the sequence of target machine instructions provides distinguishing information to distinguish between a plurality of possible outcomes for the target machine condition code, and
1[d]. directly calculates the target machine condition code without using branch instructions, the directly calculating comprising:	directly calculates the target machine condition code without using branch instructions, the directly calculating comprising:
1[e]. determining an intermediate condition code value, the intermediate condition code value being a provisional value for the target machine condition code and	determining an intermediate condition code value, the intermediate condition code value being a provisional value for the target machine condition

subject to change based on the distinguishing information; and	code and subject to change based on the distinguishing information; and
1[f]. determining, based on the intermediate condition code value and based on the distinguishing information, the target machine condition code, wherein at least part of said distinguishing information is separate from the intermediate condition code value.	determining, based on the intermediate condition code value and based on the distinguishing information, the target machine condition code, wherein at least part of said distinguishing information is separate from the intermediate condition code value.

137. The disclosure of the computer system and the instructions for execution on a processor to perform the method discussed in 1[a1] and 1[a2] similarly discloses the computer system, memory, and processor in communication with the memory, capable of performing the method recited in the first element claim 11 (see below), and the preamble of claim 16. The remaining limitations of claims 11 and 16 recite substantively identical language to limitations 1[b]-1[f], and are disclosed for the same reasons discussed above.

11. A computer system for emulating computer instructions from a source machine to	See paragraph above, §§IX.B.1-2 (1[a1]-1[a2])
--	---

produce sequences of instructions on a target machine, the computer system comprising: a memory; and a processor in communication with the memory, wherein the computer system is capable of performing a method, the method comprising:	
obtaining by the target machine a computer instruction from the source machine, said source machine having a different architecture from said target machine; and	<i>See §IX.B.3 (1[b])</i>
generating a sequence of target machine instructions which together operate to derive an encoding for a target machine condition code for the computer instruction, wherein the sequence of target machine instructions provides distinguishing information to distinguish between a plurality of possible outcomes for the target machine condition code, and directly calculates the target machine condition code without using	<i>See §§IX.B.4-5 (1[c]-[d])</i>

branch instructions, the directly calculating comprising:	
determining an intermediate condition code value, the intermediate condition code value being a provisional value for the target machine condition code and subject to change based on the distinguishing information; and	See §IX.B.6 (1[e])
determining, based on the intermediate condition code value and based on the distinguishing information, the target machine condition code, wherein at least part of said distinguishing information is separate from the intermediate condition code value.	See §IX.B.7 (1[f])

16. A method for emulating computer instructions from a source machine to produce sequences of instructions on a target machine, said method comprising::	See paragraph above, §§IX.B.1-2 (1[a1]-1[a2])
obtaining by the target machine a computer instruction from the source machine, said source	See §IX.B.3 (1[b])

machine having a different architecture from said target machine; and	
generating a sequence of target machine instructions which together operate to derive an encoding for a target machine condition code for the computer instruction, wherein the sequence of target machine instructions provides distinguishing information to distinguish between a plurality of possible outcomes for the target machine condition code, and directly calculates the target machine condition code without using branch instructions, the directly calculating comprising:	See §§IX.B.4-5 (1[c]-[d])
determining an intermediate condition code value, the intermediate condition code value being a provisional value for the target machine condition code and subject to change based on the distinguishing information; and	See §IX.B.6 (1[e])
determining, based on the intermediate condition code value and based on the	See §IX.B.7 (1[f])

distinguishing information, the target machine condition code, wherein at least part of said distinguishing information is separate from the intermediate condition code value.	
---	--

138. As explained above in §§IX.B.1-2, it would have been obvious to a POSA that a sequence of instructions (e.g., PCZ depicted as executing on the processor of target computer M2) would reside in a memory that is in communication with the processor. Processors used with the IBM/390 and the MIPS processor described in the embodiments of Loderer store program instructions (and data) in a memory that is accessible to the processor, and must do so in order to execute computer programs. Accordingly, Loderer renders obvious claims 11 and 16 for the same reasons described with respect to claim 1.

X. GROUND 2: CLAIMS 1-4, 6, 10-12, AND 16-17 ARE OBVIOUS TO LODERER IN VIEW OF POO, KANE, AND THE KNOWLEDGE OF A POSA

A. Motivation to Combine Loderer with PoO and Kane

139. Loderer relates to porting hardware-specific program code from a source hardware (e.g., IBM/390) to a target hardware (e.g., MIPS-based RISC architecture). Ex. 1005, col. 1 ¶¶1-3, col. 5 ¶3. Loderer explains “[n]ew computer generations, especially those with RISC architecture, usually have no condition code.” *Id.* Thus, transforming hardware-specific program code with condition

codes to program code without condition codes requires a simulated software level condition code. *Id.*

140. Loderer describes that an “obvious and commonly used method” for emulating condition codes is a cascade of “if” statements that query different conditions and set the appropriate condition code—i.e., building “a multi-level branch.” *Id.*, col. 1 ¶4; col. 5 ¶4. Branches, however, are implemented with jump commands in machine code “which for many computers is ‘time consuming’ relative to other machine commands, i.e., requiring a high number of processor clock cycles, results in significant loss of speed.” *Id.*

141. To avoid the disadvantages associated with branches, Loderer discloses a method to emulate condition codes at the target system without branches. Specifically, Loderer discloses a target machine command sequence that sets the condition code for an exemplary comparison instruction without using branches. Ex. 1005, Fig. 4. Loderer states:

In contrast, the method according to the invention for simulating a condition code in a code transformation does not require jumps. For the above example of a comparison instruction, FIG. 4 depicts the command sequence generated in the program code PCZ for the target hardware M2 and consisting of the subsections ZC₁, ZC₂, ZC₃ of the additional code and the commands B1 and B2.

Id., col. 6 ¶2; *see also id.*, col. 7 claim 1 (“each instruction (A_n) influencing the value (CCW) of the condition code (CC) in the program code (PCU) for the source hardware (M1) is simulated in the program code (PCZ) for the target hardware (M2) without using branching commands”).

142. Importantly, Loderer discloses the described technique is applied to **“any instruction potentially influencing the value of the condition code** is transformed by being simulated in the program code for the target hardware by means of a command generating a result value in the course of the execution of the program and without using branching commands.” Ex. 1005, col. 2 ¶1. A POSA would have been motivated by the express teachings of Loderer to emulate condition codes without branches for other IBM/390 instructions (in addition to the comparison instruction) when porting program code from the IBM/390 source machine to a MIPS-based RISC architecture machine. As Loderer explains, emulating condition codes without branches avoids time- and resource-consuming jump commands. Ex. 1005, col. 1 ¶4, col. 5 ¶4.

143. PoO, an IBM publication, discloses the full IBM/390 instruction set, including which condition codes the instructions use and the circumstances for setting each condition code. Ex. 1008, 941 (C-1). A POSA would have been familiar with PoO as a source of information regarding the IBM/390 instruction set. A POSA porting hardware-specific program code from IBM/390 source

hardware to a target MIPS-based RISC architecture hardware—e.g., as described in Loderer—would have been motivated to support the various instructions in the IBM/390 instruction set defined in the PoO that generate condition codes because Loderer specifically discloses its technique is applied to all instructions potentially influencing the value of condition codes to avoid time- and resource-consuming jump commands, and doing so would enable the instructions to be ported for more efficient execution in emulation. Ex. 1005, col. 2 ¶1. In doing so, a POSA would have been motivated by the teachings of Loderer to avoid branches when emulating condition codes for IBM/390 instructions, including for example, arithmetic functions (e.g., addition, subtraction, multiplication, division, etc.) disclosed in PoO. Ex. 1008, 316 (7-12, add); *see also id.*, 402 (7-98, multiply), 431 (7-127, subtract). The fact that an overflow may occur with a computer add or multiply instruction is such a commonly understood fact, that a POSA would have considered the possibility of overflow arising from an addition even without reference to PoO. Though a POSA would generally likely consult a reference to confirm precise behavior in a given system, a POSA reading Loderer’s description of the overflow code for the IBM/390 architecture would understand that such overflow codes would be generated when an arithmetic addition generates an overflow.

144. Loderer discloses that condition codes in the IBM/390 architecture are

used to indicate arithmetic overflows. Ex. 1005, col. 1 ¶2, col. 5 ¶3. For example, Loderer discloses the condition code value for indicating a numeric overflow. *Id.*, col. 5 ¶3 (“the value CCW = 3 is defined for the condition code CC or the associated bit pattern ‘0001’ to indicate a numeric overflow.”).

Comparison result	CCW	BM
OP1 equal to OP2	0	1000
OP1 < OP2	1	0100
OP1 > OP2	2	0010
Overflow	3	0001

FIG. 2

Id., Fig. 2.

145. Loderer does not discuss numeric overflow in detail because the exemplary comparison instruction that compares two numerical values does not implicate the overflow code: “However, in comparison operations in which both operands (numerical values) are well-defined, an overflow does not matter, so that it will not be discussed in more detail below.” Ex. 1005, col. 5 ¶3. However, a POSA would have understood that numeric overflow can occur with other instructions, such as arithmetic functions.

146. For other functions (e.g., arithmetic functions) in the IBM/390 instruction set, the condition code value may indicate a numeric overflow. For example, Fig. C-1 reproduced in-part below, shows the condition code values for the IBM/390 ADD instruction (top highlighted row). A condition code of “3”

indicates a numeric overflow.

Instruction	Condition Code			
	0	1	2	3
ADD (gen)	Zero	< zero	> zero	Overflow
ADD (BFP)	Zero	< zero	> zero	NaN
ADD DECIMAL	Zero	< zero	> zero	Overflow
ADD HALFWORD	Zero	< zero	> zero	Overflow
ADD HALFWORD IMMEDIATE	Zero	< zero	> zero	Overflow
ADD LOGICAL	Zero, no carry	Not zero, no carry	Zero, carry	Not zero, carry
ADD LOGICAL WITH CARRY	Zero, no carry	Not zero, no carry	Zero, carry	Not zero, carry
ADD NORMALIZED	Zero	< zero	> zero	--
ADD UNNORMALIZED	Zero	< zero	> zero	--
AND	Zero	Not zero	--	--
CANCEL SUBCHANNEL	Function initiated	--	--	Not operational
CHECKSUM	Checksum complete	--	--	CPU-determined completion
CIPHER MESSAGE	Normal completion	--	--	Partial completion
CIPHER MESSAGE WITH CHAINING	Normal completion	--	--	Partial completion
CLEAR SUBCHANNEL	Function initiated	--	--	Not operational
COMPARE (gen, HFP)	Equal	Low	High	--
COMPARE (BFP)	Equal	Low	High	Unordered
COMPARE AND FORM CODEWORD	Equal	OCB=0: low OCB=1: high	OCB=0: high OCB=1: low	--
COMPARE AND SIGNAL	Equal	Low	High	Unordered
COMPARE AND SWAP	Equal	Not equal	--	--

Figure C-1 (Part 1 of 6). Summary of Condition-Code Settings

Ex. 1008, 941 (C-1).

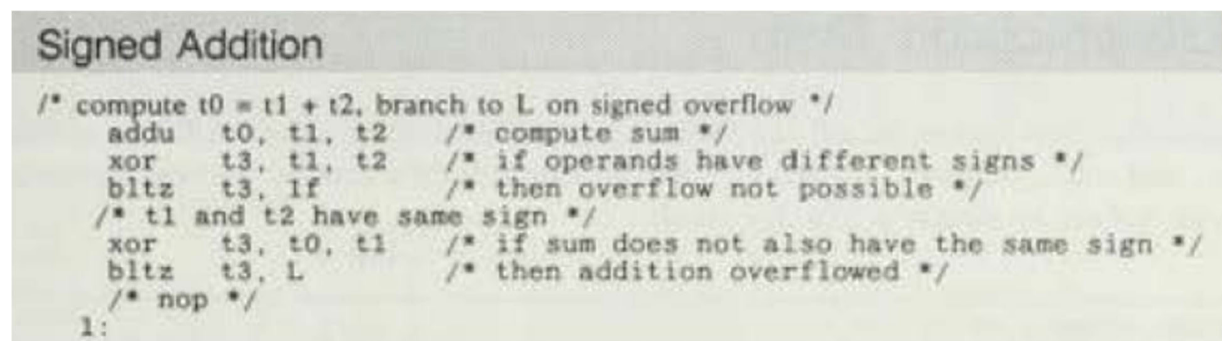
147. A POSA would have been motivated to accurately emulate condition codes for all IBM/390 instructions so the resulting program code on the MIPS-based RISC architecture target machine would function correctly. For example, instructions that rely on the simulated condition code may not operate functionally the same on the target machine if a numeric overflow is not detected and identified with a simulated condition code. In such a scenario, the undetected overflow may cause the resulting program code to operate in an unexpected manner. For

example, if two values are added together and the sum is too large to fit in a register (e.g., a numeric overflow condition), the value stored in the result register may not accurately reflect the sum of the two values. Without detecting and identifying the numeric overflow with a simulated condition code, later instructions may rely on the inaccurate sum value if the overflow is not detected and identified. This may result in unexpected behavior and errors in the resulting program code, some of which may be difficult to detect and diagnose. A POSA would consider arithmetic operations when considering overflow because these instructions are capable of generating an overflow.

148. Though Loderer and PoO do not disclose how to detect an arithmetic overflow, detecting an overflow condition was obvious and well within the skill of a POSA. A given supported format for storing numbers and performing arithmetic operations on those numbers uses a defined number of bits to store the numbers used in the calculations. A POSA would understand that addition on any system supporting such a format can produce a sum whose value is larger than the available number of bits, and that this circumstance is known as an “overflow.” The possibility of an overflow, and techniques for detecting and appropriately handling the overflow case, are well-known problems that have been addressed in any correctly-functioning computer architecture, and a POSA would know that references relating to a particular architecture would address the overflow case. For

example, Kane describes the MIPS instruction set architecture, including the behavior of instructions, and how to program an R2000 machine with a MIPS processor using MIPS instructions. Ex. 1009. “The book is a comprehensive reference manual for the MIPS RISC architecture.” Ex. 1009, 49 (back cover of book). A POSA porting program code to the MIPS RISC architecture, as Loderer discloses, would have consulted MIPS-related materials, such as Kane.

149. Kane includes a section on testing whether an arithmetic operation resulted in an overflow. Ex. 1009, 45 (C-8, “Testing for Overflow”). Kane discloses specific MIPS instruction sequences for detecting overflow for various arithmetic functions. *Id.*, 46 (C-9). For example, Kane disclose a two-step MIPS instruction sequence for detecting an arithmetic overflow for signed addition and setting a condition code to indicate an overflow occurred. Ex. 1009, 45 (C-8), 46 (C-9).



The image shows a snippet of MIPS assembly code titled "Signed Addition". The code is as follows:

```
/* compute t0 = t1 + t2, branch to L on signed overflow */
addu  t0, t1, t2    /* compute sum */
xor   t3, t1, t2    /* if operands have different signs */
bltz  t3, if        /* then overflow not possible */
/* t1 and t2 have same sign */
xor   t3, t0, t1    /* if sum does not also have the same sign */
bltz  t3, L        /* then addition overflowed */
/* nop */
L:
```

I understand that Kane is a well-known resource for MIPS programmers before the '289 patent as evidenced by the hundreds of citations to the book. Kane thus discloses, and confirms a POSA would understand, how to detect arithmetic

overflow in a MIPS RISC architecture and would have known or been motivated to consider the instruction sequences disclosed in Kane for detecting arithmetic overflow.

150. The instruction sequences for detecting overflow disclosed in Kane use branches (the “bltz” instructions above), which as Loderer explains was “[a]n obvious and commonly used method.” Ex. 1005, col. 1 ¶4. As explained above, a POSA would have been motivated by the express teachings of Loderer to emulate condition codes on the MIPS-based target machine without branches (e.g., as Loderer discloses for a comparison instruction) to avoid the disadvantages caused by the branches disclosed by Kane.

151. In view of Loderer, PoO, Kane, and the knowledge of a POSA, a POSA would have been motivated to emulate a condition code for an overflow condition without using branch instructions, and would have had a reasonable expectation of success in doing so. For example, a POSA would have been motivated to detect an arithmetic overflow for the ADD instruction (using the technique taught by Kane) and generate an overflow code (as disclosed in PoO and Loderer) without using branch instructions (as taught by Loderer). Below I illustrate why a POSA would have reasonably expected success in using Loderer’s technique of storing a “1” or a “0” in a register based on comparisons to do so.

B. A POSA Would Have Had a Reasonable Expectation of Success Applying Loderer's Technique to Calculate an Overflow Code for an ADD Instruction Without Branches

152. The ADD instruction is an exemplary function that illustrates how a POSA would reasonably expect success in (i) detecting numeric overflow (as disclosed by Kane) and (ii) setting a condition code to reflect the overflow (as disclosed by PoO and Loderer), both without branches (as taught by Loderer). Other instructions that set overflow condition codes could be similarly implemented to render the claims in this ground obvious. The below discussion regarding the ADD instruction confirms a POSA would have had a reasonable expectation of success detecting the overflow condition and calculating the overflow code without using branch instructions. These implementations were well within the skillset of a POSA before the '298 patent.

1. Exemplary ADD Instruction Implementation

153. As explained in §VII.A and ground 1 above, Loderer discloses a sequence of instructions to calculate the condition code for a compare instruction, which PoO discloses results in the same condition codes for the non-overflow case of an ADD instruction. If sum=0 (BM=1000), sum>0 (BM=0010), or sum<0 (BM=0100). Ex. 1008, 941 (C-1); Ex. 1005, Figs. 2, 4.

Command sequence in PCZ		a = 0	a > 0	a < 0
ZC ₁	Set BM to '0100'	BM = 0100	BM = 0100	BM = 0100
B1	If a = 0, set t ₁ = 1, otherwise t ₁ = 0	t ₁ = 1	t ₁ = 0	t ₁ = 0
ZC ₂	BM << t ₁	BM = 1000	BM = 0100	BM = 0100
B2	If a > 0, set t ₂ = 1, otherwise t ₂ = 0	t ₂ = 0	t ₂ = 1	t ₂ = 0
ZC ₃	BM >> t ₂	BM = 1000	BM = 0010	BM = 0100

FIG. 4

Ex. 1005, Fig. 4. In the event that an ADD does not overflow, the Loderer instructions above (that correspond to the comparison instruction) would correctly calculate the condition codes for the ADD instruction, and the resulting MIPS code would function correctly. This is true because, in the non-overflow case, the ADD instruction sets the same condition codes the COMPARE instruction does when comparing an operand to zero. Loderer Figs. 2-4 show the codes set when the COMPARE instruction compares an operand to zero: code 0 for sum=0, code 1 for sum<0, and code 2 for sum>0. PoO shows that the same condition codes are used for the same three cases with the ADD instruction:

Instruction	Condition Code			
	0	1	2	3
ADD (gen)	Zero	< zero	> zero	Overflow
ADD (BFP)	Zero	< zero	> zero	NaN
ADD DECIMAL	Zero	< zero	> zero	Overflow
ADD HALFWORD	Zero	< zero	> zero	Overflow
ADD HALFWORD IMMEDIATE	Zero	< zero	> zero	Overflow
ADD LOGICAL	Zero, no carry	Not zero, no carry	Zero, carry	Not zero, carry
ADD LOGICAL WITH CARRY	Zero, no carry	Not zero, no carry	Zero, carry	Not zero, carry
ADD NORMALIZED	Zero	< zero	> zero	--
ADD UNNORMALIZED	Zero	< zero	> zero	--
AND	Zero	Not zero	--	--
CANCEL SUBCHANNEL	Function initiated	--	--	Not operational
CHECKSUM	Checksum complete	--	--	CPU-determined completion
CIPHER MESSAGE	Normal completion	--	--	Partial completion
CIPHER MESSAGE WITH CHAINING	Normal completion	--	--	Partial completion
CLEAR SUBCHANNEL	Function initiated	--	--	Not operational
COMPARE (gen, HFP)	Equal	Low	High	--
COMPARE (BFP)	Equal	Low	High	Unordered
COMPARE AND FORM CODEWORD	Equal	OCB=0: low OCB=1: high	OCB=0: high OCB=1: low	--
COMPARE AND SIGNAL	Equal	Low	High	Unordered
COMPARE AND SWAP	Equal	Not equal	--	--

Figure C-1 (Part 1 of 6). Summary of Condition-Code Settings

Ex. 1008, 941 (C-1).

154. In the event of an overflow, BM should be changed to “0001” (condition code 3) as defined in PoO and disclosed by Loderer. Ex. 1008, 941 (C-1, overflow corresponds with condition code 3); Ex. 1005, Fig. 2 (condition code 3 corresponds with BM “0001”). Though one could easily make this change to the value of BM using a branch instruction, Loderer specifically teaches that a POSA should avoid the cascade of branches, making use of the technique of conditionally setting a register without branching, and adjusting the value of the condition code accordingly. The sections below illustrate detecting the overflow (as taught by

Kane) and using an overflow code calculated without branch instructions (using Loderer's techniques), which demonstrate a POSA's reasonable expectation of success in the identified combination.

2. Detecting the overflow

155. Kane discloses techniques for detecting overflow without traps, which is relevant here for detecting overflow without branches. Ex. 1009, 45 (C-8), 46 (C-9), 13 (5-18, "R2000 branches to the General Exception vector" for overflow exception). For signed addition, Kane explains that "overflow occurs if the signs of the addends are the same and the sign of the sum is different." Ex. 1009, 45 (C-8). A POSA would recognize that this is the general test for a two's complement overflow, and would apply to any computer that performs two's complement addition—which is ubiquitous in computing. For the overflow to occur, two conditions must be satisfied:

overflow = (addends same sign) **AND** (sum different sign).

For example, if the result of summing two positive addends is negative, which should not occur, an overflow is detected. For a MIPS processor, overflow detection is handled in software. Ex. 1009, 43 (C-1), 45 (C-8). Kane's exemplary code uses bitwise exclusive OR (XOR) instructions to test for each of these conditions related to overflow detection.

Signed Addition

```

/* compute t0 = t1 + t2, branch to L on signed overflow */
addu  t0, t1, t2    /* compute sum */
xor   t3, t1, t2    /* if operands have different signs */
bltz  t3, if        /* then overflow not possible */
/* t1 and t2 have same sign */
xor   t3, t0, t1    /* if sum does not also have the same sign */
bltz  t3, L        /* then addition overflowed */
/* nop */
L:

```

Ex. 1009, 45 (C-8), 46 (C-9). An XOR is a logical operation that returns true (“1”) if its arguments differ and returns false (“0”) if its arguments are the same. The MIPS “xor” instruction shown above is a bitwise instruction, such that it compares each bit of register t1 to the corresponding bit of register t0, storing a “1” result bit in the t3 bit location if the bits are different and a “0” otherwise. *Id.*, 41 (A-85, XOR instruction). Below is an example of a bitwise xor on a four-bit binary number:

```

    0101
XOR 0011
    = 0110

```

156. The addu instruction in Kane’s algorithm stores the sum of the two addends (t1 and t2) in register t0. “Note that the only difference between this [ADDU] instruction and the ADD instruction is that ADDU never causes an overflow exception.” Ex. 1009, 24 (A-12). Thus, the addu instruction operates on two signed register operands to produce a signed result without generating an overflow exception. The first xor instruction in Kane tests whether the addends have different or the same sign, storing a negative number (i.e., a binary number with a leading “1”) in register t3 if the signs differ. Like many computers, the MIPS

processor stores signed numbers in two's complement form, which means that a negative number has a "1" in its leading bit, whereas a positive number has a "0" in its leading bit. Thus, if the addends (i.e., t1 and t2) have different signs, then t3 (the result of the first xor instruction) will have a "1" in its leading bit, representing a negative number in t3. If the addends have the same signs, then t3 will have a "0" in its leading bit, representing a positive number in t3. Kane's exemplary code uses a branch-less-than-zero instruction, (bltz t3, 1f) that branches when t3 is less than zero, which corresponds to the addends having different signs such that overflow is not possible. Thus, Kane's exemplary code branches to 1f if overflow is not possible.

157. If the signs of the addends are the same (indicating an overflow is possible and the first branch did not happen), Kane's second xor statement tests the second condition (whether the sign of the sum t0 differs from that of the first addend t1), storing a negative number in register t3 if the signs differ which indicates an overflow. Ex. 1009, 45 (C-8), 46 (C-9).

158. The second branch-less-than-zero instruction, (bltz t3, L) branches when t3 is less than zero, which corresponds to the addends have different signs such that overflow is not possible. In other words, Kane detects negative results of the second xor instruction and branches to L if a negative result is detected (i.e., when an overflow occurs).

159. A POSA would have been motivated to apply Loderer's techniques to implement Kane's two-step overflow detection without branching. Loderer expressly discloses avoiding jump commands "which for many computers is 'time-consuming' relative to other machine commands." Ex. 1005, col. 1 ¶4; col. 5 ¶4. Loderer also discloses using machine instructions that store information about a comparison in a register to avoid the need to branch, and Loderer suggests the same technique could be used to simulate the overflow condition. Ex. 1005, Figs. 2, 4. A POSA would reasonably expect success in detecting overflow without branch instructions, and would understand at least three alternative implementations for doing so: (i) using conventional bit manipulation (bitwise) instructions of the MIPS architecture, (ii) by using the instructions described by Loderer for comparing and setting a register, or (iii) by using the slt MIPS instruction, as described below:

Detecting overflow using bitwise instructions

160. Because the first bit (the sign bit in two's complement) produced by each of Kane's xor instructions contains the information as to the respective steps of Kane's test for overflow, a POSA would have understood that simple bit manipulations would store a "1" in a register for each prong of the test. A POSA would know this because a bitwise xor performs the logical xor operation on corresponding bits of the two arguments. The first bit of the result contains the xor

of the first bit of each argument. The second bit of the result contains the xor of the second bit of each argument, and so on. In the case of the signed binary arguments of the AND instruction, the first bit of each number denotes the sign, with a leading “1” indicating a negative number. Below is an example of using bitwise instructions (highlighted below) to achieve the same result of Kane without branches.

161. It would be obvious to a POSA to extract the sign information from the result of the xor instruction and to use this information to detect the signs and changes in sign. In instruction (3a) below, the shift right logical (SRL) MIPS instruction moves the sign bit of t6 to the right and can be used to shift the sign bit 31 places, resulting in a “1” if the signs were different and a “0” if the signs were the same. Ex. 1009, 39 (A-70). Shifting a number to move a particular bit in a value to a desired position is a routine manipulation, and it would be obvious to a POSA to use the SRL instruction to do so. Having shifted the value into the right-most bit, a “1” would mean that overflow is not possible. Inverting the bit would create an indicator that overflow is possible, and the nor instruction (4a) inverts the bit, so a “1” in t6 indicates that an overflow is possible. *Id.*, 33 (A-56).

1a. addu a, t4, t5

2a. xor t6, t4, t5

3a. SRL t6, t6, 31

4a. nor t6, t6, \$0

In an instruction set that provided a “not” instruction, it would have been obvious to use the available “not” instruction. A POSA would understand that a logical NOR operation will produce the same result as a NOT instruction if the argument is NOR’ed with a zero. Using pseudocode, the ’289 patent describes the use of an instruction called “flip_bit” to invert a bit in a specific register position and to store the result in a target register. Ex. 1001, 5:12-14. Pseudocode is used to communicate a way in which the code would operate, without restricting the description to a machine’s specific syntax. *Id.*, 4:41-48. The use of pseudocode by the patent illustrates that recognizing machine instructions that may be used to isolate a bit of interest and to invert it using the instructions available in a given processor’s instruction set architecture is within the level of skill in the art. In this case, a POSA would know that the bitwise nor instruction, which is available on the MIPS architecture, can be used to invert the bit, by NORing the register with the value zero. MIPS also includes the \$0 register, which operates as a register whose value is always zero. Ex. 1009, 42 (D-2, table D.1). The value zero is commonly used in computing, and some instruction set architectures provide a zero value without requiring the programmer to set or clear a general register to zero. To implement the same logic without using a fixed zero-value register, a POSA would make use of one or another known technique to set a zero-value into a register. Moving an immediate

value of zero into a register, or using other bitwise operations (XORing a value with itself, and using ANDI to AND the contents of a register with the immediate value zero), for example, are known ways to set a register to zero. Applying the same shift technique (instruction (6a)) after the second xor instruction (5a) puts a “1” in t7 if the sign of the sum (a) differs from the sign of the first addend (t4).

5a. xor t7, a, t4

6a. SRL t7, t7, 31

If the signs of the two addends were the same (e.g., if t6=1), then t7=1 indicates an overflow. However, because the above sequence eliminates Kane’s branch instructions, it is possible that t7=1 because the sum (a) was of different sign than the first addend, but was not different from both addends because the addends were not the same sign (t6=0). A POSA would understand a bitwise AND of t6 and t7 (instruction (7)) effectively checks that both Kane conditions for an overflow are present (overflow = (addends same sign) **AND** (sum different sign)), storing a “1” in t3 if an overflow exists.

7a. and t3, t6, t7

162. Below is a side-by-side comparison of the original Kane instructions with the modifications discussed above that illustrate the modified instructions use the addu and xor instructions similar to Kane to detect whether an overflow is possible, and whether an overflow has occurred:

Kane instructions	Modified instructions without branches
addu t0, t1, t2	addu a, t4, t5
xor t3, t1, t2	xor t6, t4, t5
bltz t3, 1f	SRL t6, t6, 31
	nor t6, t6, \$0
xor t3, t0, t1	xor t7, a, t4
bltz t3, L	SRL t7, t7, 31
	and t3, t6, t7

Where Kane used branches (i.e., bltz) to detect and signal an overflow, the modified version uses bitwise manipulations with an AND instruction. Ex. 1009, 25 (A-13), 26 (A-22).

163. Thus, using Loderer's technique of storing a "1" or a "0" in a register based on comparisons to generate an overflow code, a POSA would be motivated by Loderer to detect an arithmetic overflow as taught by Kane without using undesired branch instructions. The above use of bitwise and Boolean logic to extract and combine bits to put a signal in the desired form is a routine programming practice that would be known to a POSA.

Detecting overflow using Loderer's instructions

164. As an alternative to the bitwise instructions discussed above, a POSA could use Loderer's combined instructions, e.g., commands B1 and B2 in Fig. 4.

Loderer states: “To simulate an instruction, it may also be necessary to combine several, even different ones, of such result value-generating commands.” Ex. 1005, col. 2 ¶2. It would have been obvious to a POSA that following the first xor (instruction 2b taught by Kane), one could combine Loderer’s commands B1 and B2 (instructions (3b-4b) below) to generate an indicator that the addends were of the same sign. Instruction (3b) sets t7=1 if the addends are the same value (and thus the same sign), and instruction (4b) sets t8=1 if the addends are of the same sign:

1b. addu a, t4, t5

2b. xor t6, t4, t5

3b. If t6=0, set t7=1, otherwise t7 =0

4b. If t6 >0, set t8=1, otherwise t8 = 0

5b. or t7, t7, t8

Because we want to detect whether t6 is greater than OR equal to zero, the OR instruction (5b) logically combines the results of t7 and t8 and stores the result in t7, such that if t7=0 after instruction (5b), an overflow is not possible (the first step of Kane’s two-step overflow detection). Ex. 1009, 34 (A-57).

165. The same technique sets register t9 to zero if the sign of the sum differs from the sign of the first addend (instructions (7b-9b) below):

6b. xor t8, a, t4

7b. if t8 = 0, set t9=1 else t9=0

8b. if $t8 > 0$, set $t10=1$ else $t10=0$

9b. or $t9$, $t9$, $t10$

10b. nor $t3$, $t9$, $\$0$

Because a POSA would understand a “1,” rather than a zero to indicate the overflow in the context of Kane’s algorithm, the nor instruction (10b) inverts the bit. Ex. 1009, 33 (A-56). As in the sequence that relied on the shift instructions, a POSA would understand that combining the results of the two tests together with an AND instruction (11b) produces an overflow signal)), storing a “1” in $t3$ if an overflow exists.

11b. and $t3$, $t3$, $t7$

166. The use of instructions to set and manipulate bits in registers is a routine programming task. In this case, the nor instruction operates to invert bits in the same manner discussed in the previous example.

Detecting overflow using the MIPS slt instruction

167. In the example that uses the Loderer’s instructions for performing a comparison and setting a register, the two instructions were used and combined with an OR instruction in order to detect whether $t6$ was greater than or equal to zero. Because the MIPS architecture includes an instruction that performs a comparison (less than) and sets a register based on whether or not the argument is less than zero (set on less than, SLT), it would be obvious to a POSA that one

could use the SLT instruction in lieu of the two Loderer instructions to detect the overflow. A POSA would understand the Loderer instructions (3b-4b) in the previous example implementation could be substituted to with the slt instruction to achieve the same result:

1c. addu a, t4, t5

2c. xor t6, t4, t5

3c. slt t7, t6, \$0

The slt instruction (3c) sets t7=1, if t6<0 which represents an overflow is possible (the first step of Kane's two-step overflow detection) or t7=0 otherwise. Ex. 1009, 38 (A-64, slt instruction). Thus, an overflow is possible if t7=1, but not if t7=0.

168. Testing whether the value of a parameter is less than a threshold value ($a < \text{threshold}$), or testing whether the parameter is greater than or equal to a threshold value ($a \geq \text{threshold}$) and inverting the result are logically equivalent. Identifying different available instructions to perform logically equivalent operations is a routine programming task.

169. The same technique sets register t8=0 if the sign of the sum differs from the sign of the first addend (instructions 4c-5c below):

4c. xor t8, a, t4

5c. slt t8, t8, \$0

The slt instruction (5c) sets t8=1, if t8<0 which represents an overflow is possible (the second step of Kane's two-step overflow detection) or t8=0 otherwise. Ex. 1009,

38 (A-64, slt instruction). Thus, an overflow is possible if t8=1, but not if t8=0. As in the examples above, a POSA would understand that combining the results of the two tests together with an AND instruction (6c) would produce an overflow signal, storing a “1” in t3 if an overflow exists.

6c. and t₃, t₇, t₈

170. The exemplary sequences above (instructions 1a-7a, 1b-10b, 1c-6c) are two examples showing how, applying Loderer’s direction to calculate condition codes without branching by performing comparisons and then setting register values containing the information obtained by the comparison, a POSA would reasonably expect success in detecting overflow and calculating the overflow code without branch instructions using the overflow detection techniques disclosed in Kane.

3. Calculating the overflow condition code

171. As explained in §§X.A-B, it would have been obvious to a POSA to calculate the overflow code without using branch instructions, and to store the overflow code in a register, such as register t3.

172. Using the overflow code, it would have been obvious to a POSA, in view of Loderer’s technique of shifting and modifying the BM register to produce condition codes for the COMPARE instruction, to apply shifts and arithmetic operations to the same BM register to address the overflow code, as shown below.

Command sequence in PCZ		a = 0	a > 0	a < 0	Overflow
ZC ₁	Set BM to '0100'	BM = 0100	BM = 0100	BM = 0100	BM = 0100
B1	If a = 0, set t ₁ = 1, otherwise t ₁ = 0	t ₁ = 1	t ₁ = 0	t ₁ = 0	-
ZC ₂	BM << t ₁	BM = 1000	BM = 0100	BM = 0100	-
B2	If a > 0, set t ₂ = 1, otherwise t ₂ = 0	t ₂ = 0	t ₂ = 1	t ₂ = 0	-
ZC ₃	BM >> t ₂	BM = 1000	BM = 0010	BM = 0100	-
ZC ₂	BM << t ₃	BM = 1000	BM = 0010	BM = 0100	BM = xx00
ZC ₂	BM << t ₃	BM = 1000	BM = 0010	BM = 0100	BM = x000
ZC ₂	BM << t ₃	BM = 1000	BM = 0010	BM = 0100	BM = 0000
A _n	addu BM, BM, t ₃	BM = 1000	BM = 0010	BM = 0100	BM = 0001

Ex. 1005, Fig. 4 (modifications in yellow).

173. In the overflow case (t₃=1), the three highlighted shift instructions shift zeros into the bits in the BM register, producing interim value BM=0000. If there is no overflow (t₃=0), the shift instructions do not change the value of BM because each of the highlighted ZC₂ instructions shown above will shift BM by 0 bits. Using shifts to selectively change, or not change, the value of BM is precisely how Loderer applies the distinguishing information in t₂ and t₁ to adjust the value of BM for the non-overflow case. It would have been obvious to a POSA to employ the same approach to selectively shift BM to apply the distinguishing information (t₃) in the overflow case. The final instruction in the table above adds t₃ (the overflow indicator) to BM. In the case of an overflow, the sum of zero (0000) and one (0001) produces a result of 0001 in BM, which is the correct code for the overflow case. Otherwise t₃=0 and neither the shifts nor the addition change the value of the condition code, leaving the correctly computed BM values for a=0, a<0, or a>0 in place.

174. As explained in §X.A and §X.B, a POSA would have been motivated to combine Loderer with PoO and Kane and had a reasonable expectation of success of doing so to generate the condition codes for the ADD instruction. As shown above, detection of the overflow, and storing the detected overflow as an overflow code in a register (e.g., t_3) would have been obvious in view of Loderer's direction and Kane's teaching of the two steps for detecting an overflow. Either of the illustrative code sequences would detect the overflow code.

175. Once the overflow code was available, a POSA would have been motivated by Loderer, to extend Loderer's technique of using shift instructions and the overflow code stored in t_3 , to emulate the condition codes for the IBM/390 ADD instruction on a MIPS processor, without using branch instructions.

C. Claim 1

176. As explained in §X.A and §X.B it would have been obvious to a POSA to extend Loderer, in view of PoO and Kane to directly calculate emulated condition codes without branch instructions for the IBM/390 ADD instruction, and to include the same code sequence discussed and analyzed in ground 1 for the COMPARE instruction, to generate the codes for the non-overflow case. This extension of Loderer includes calculating the overflow code, as detailed in §X.B, and applying the overflow code to modify the condition code, as set forth in §X.B, in the overflow case. When there is no overflow, the value of the overflow code

t3=0, and the extended code does not alter the condition codes generated by Loderer's code, as set forth in Loderer Fig. 4 for the COMPARE instruction. Thus, it would be obvious to a POSA that Loderer, as modified in the description of §X.B, renders claim 1 obvious for at least the reasons described in ground 1. *See* §IX.B.

177. The modification of Loderer detailed in §X.B above includes a third example of “distinguishing information”—namely the overflow code stored in register t3. When an IBM/390 ADD instruction is received and the setting of its condition codes emulated as described in §X.A and §X.B above, limitations 1[a1], 1[a2], and 1[b] are satisfied in the same manner as in the analysis for ground 1. The instructions are received from the source machine (M1) and the instruction sequence to emulate generating the condition code, three examples of which are detailed in §X.B above, executes on the target machine (M2). For the non-overflow case, the remaining limitations of claim 1 are also satisfied in the manner described in ground 1. In the overflow case, additional distinguishing information is used as described below.

178. This third bit of distinguishing information (information distinguishing overflow from non-overflow) is generated by the code sequences described in §X.B, thus rendering obvious limitation 1[c] in the overflow case. The value of t3 is used to change a provisional value of the condition code into the final

value for the overflow case (BM=0001), as described in §X.B.3, thus rendering obvious limitation 1[d]. In the overflow case, the values of BM calculated by Loderer's code sequence of Fig. 4 produces an interim value of the condition code, BM, that is subject to change based on the value of the overflow code (t3), thus rendering obvious limitation 1[e]. See §X.B.3.

179. Like the distinguishing information stored in registers t1 and t2 discussed in ground 1, the additional distinguishing information stored in register t3 resides outside of the condition code itself, thus disclosing limitation 1[f].

180. Thus, Loderer as extended to calculate the overflow condition code as required for the ADD instruction, in view of PoO and Kane, renders claim 1 obvious for these additional reasons.

D. Claim 2: The computer program product of claim 1, wherein said calculation of the target machine condition code comprises using at least one of carry, sign, and overflow codes in the calculation.

181. As explained in detail in §X.A, a POSA would have been motivated to extend Loderer, in view of PoO and Kane, to calculate condition codes for other IBM/390 instructions, such as the IBM/390 ADD instruction. As described in §X.A and §X.B, it would have been obvious to a POSA to calculate the overflow code (e.g., the detected overflow indicator t3), and then to use the calculated overflow code to calculate the target machine code (BM=0001) in the overflow case. For the non-overflow cases, the calculation of the condition codes for the

IBM/390 ADD instruction requires the same logic as the COMPARE instruction, as analyzed in detail in ground 1. Ex. 1008, 941 (C-1).

182. Loderer describes generating emulated condition codes for IBM/390 instructions on a MIPS processor, which does not have hardware flags for the condition code. Ex. 1008, 941 (C-1); Ex. 1005, col. 1 ¶¶1-3, col. 5 ¶3. As explained in detail in §X.A, it would have been obvious to a POSA, applying Loderer's technique to the overflow detection steps described in Kane, to calculate an overflow code that indicates whether or not an overflow has occurred, in order to correctly set the condition codes for the overflow case. The examples detailed in §X.B detect the overflow, calculate the overflow code, and store it in register t3.

183. As explained above, a POSA would recognize that for the non-overflow case, the sequence of instructions described in Loderer's Fig. 4 for the COMPARE instruction also calculates the correct condition codes for the non-overflow case of the ADD instruction. It would have been obvious to a POSA to apply Loderer's techniques for manipulating the condition code, based on the value of the overflow code, to change the value of the condition code to BM=0001 in the event of an overflow (t3=1), and to leave the condition code undisturbed in the event of no overflow (t3=0).

184. The example described in detail in §X.B.3, reproduced below, applies the overflow code (t3) calculated per the techniques in §X.B.2, to calculate the

target condition code for the ADD instruction.

Command sequence in PCZ		a = 0	a > 0	a < 0	Overflow
ZC ₁	Set BM to '0100'	BM = 0100	BM = 0100	BM = 0100	BM = 0100
B1	If a = 0, set t ₁ = 1, otherwise t ₁ = 0	t ₁ = 1	t ₁ = 0	t ₁ = 0	-
ZC ₂	BM << t ₁	BM = 1000	BM = 0100	BM = 0100	-
B2	If a > 0, set t ₂ = 1, otherwise t ₂ = 0	t ₂ = 0	t ₂ = 1	t ₂ = 0	-
ZC ₃	BM >> t ₂	BM = 1000	BM = 0010	BM = 0100	-
ZC ₂	BM << t ₃	BM = 1000	BM = 0010	BM = 0100	BM = xx00
ZC ₂	BM << t ₃	BM = 1000	BM = 0010	BM = 0100	BM = x000
ZC ₂	BM << t ₃	BM = 1000	BM = 0010	BM = 0100	BM = 0000
A _n	addu BM, BM, t ₃	BM = 1000	BM = 0010	BM = 0100	BM = 0001

If there is no overflow, then the three shift instructions and the addu instruction leave the value of BM unchanged. But, if t₃=1, then they operate to change the emulated condition code to the correct value for the overflow case, BM=0001. Thus, the added ZC₂ instructions selectively shift the intermediate value of the condition code stored in register BM based on the value of t₃ in the same way that Loderer selectively shifted the intermediate values in response to the distinguishing information stored in t₁ or t₂.

185. The logical add example described in the '289 patent similarly uses the distinguishing information (in this case the value stored in "rX," to set the least significant bit of the condition code to "1," and, in the "no carry case," depicted in the two left sets of columns of the table below, this has the effect of changing the value of the condition code to the value of the calculated distinguishing information. Ex. 1001, 7:19-48.

[1] add_set_carry rT, rA, rB [2] move_from_carry rC [3] double rC [4] set_bit_on_not_zero rX, rT [5] or rC, rC, rX																
rT is zero no carry cc = 0 Register Contents				rT not zero no carry cc = 1 Register Contents				rT is zero carry cc = 2 Register Contents				rT not zero carry cc = 3 Register Contents				
rC	rT	rX	c	rC	rT	rX	c	rC	rT	rX	c	rC	rT	rX	c	
[1]	n/a	0	n/a	0	n/a	not 0	n/a	0	n/a	0	n/a	1	n/a	not 0	n/a	1
[2]	0	0	n/a	0	0	not 0	n/a	0	1	0	n/a	1	1	not 0	n/a	1
[3]	0	0	n/a	0	0	not 0	n/a	0	2	0	n/a	1	2	not 0	n/a	1
[4]	0	0	0	0	0	not 0	1	0	2	0	0	1	2	not 0	1	1
[5]	0	0	0	0	1	not 0	1	0	2	0	0	1	3	not 0	1	1

Id., 7:29-48. In this example in the '289 patent, after the third instruction, the intermediate value of the condition code is either set to “0” (in binary, “00”), or two “2” (in binary, “10”). In all four cases, the value of the least significant bit of rC is zero. The fifth instruction is a bitwise OR that sets the least significant bit of rC to “1” if the distinguishing information rX=1, and leaves the bit at “0” if rX=0. In the “no carry” cases, depicted in the two left sets of columns of the table, the intermediate value of the condition code stored in rC had been zero, so the final calculated value becomes the value stored in rX.

186. Also explained in detail in §X.A and §X.B, it would have been obvious to a POSA, in view of Loderer’s teaching to generate condition codes for IBM/390 instructions, including those that signal overflows, to detect the overflow and to store the detected overflow code in a register. As explained in §X.A, it

would have been obvious for the code sequence that generates the overflow code to be written without using branch instructions, as taught by Loderer, in view of Kane.

187. The above sequences also make use of the sign bit. The addu instruction generates a signed result whose first bit is the sign bit. As explained in §X.B.2, the sign bit is used to determine whether the sign of the result differed from the sign of the addends. Thus, it would not only be obvious to a POSA to use the calculated overflow code (t3), but also to use the sign bit (first bit of the computed sum), in calculating the target machine condition code.

188. Thus, the combination of Loderer, PoO, Kane, and the knowledge of a POSA renders claim 2 obvious.

E. Claim 3: The computer program product of claim 2, wherein said calculation employs temporary locations for storing intermediate results.

189. As explained in §VII.A. §IX.A, and 1[e] in ground 1, the '289 patent discloses the use of a register, rC (a “temporary location”), storing an interim value of a condition code, where the interim value may be changed, depending on the value of the distinguishing information stored in register rX. The '289 patent refers to rX, rY, and rZ as “temporary registers used to hold intermediate results.” Ex. 1001, 5:24-25.

190. Similar to the '289 patent, Loderer employs temporary locations for

storing intermediate results related to calculating the condition code. As explained in §X.B and §X.C with respect to claim 1, the extension of Loderer stores intermediate results in the BM register, which is a temporary location. Ex. 1005, col. 6 ¶¶4-5, Fig. 4. For example, loading the initial value of BM with instruction ZC_1 stores an intermediate condition code value in a temporary register BM. *Id.* The interim value of the condition code stored in register BM may be changed, based on the value of the distinguishing information. *Id.*

191. Loderer, as modified in §X.A and §X.B, also discloses using additional temporary locations for calculating the condition code. For example, the temporary registers (e.g., t0, t1, t2, t3, t4, t5, t6, t7, t8, and/or t9) used for detecting an overflow and setting the overflow condition code as detailed in §X.B.2, are also examples of temporary locations for storing intermediate results of the condition code instructions, similar to the registered disclosed in the '289 patent (e.g., rX, rY, and rZ).

192. Thus, the combination of Loderer, PoO, Kane, and the knowledge of a POSA renders claim 3 obvious.

F. Claim 4: The computer program product of claim 1, wherein said method further includes executing said generated sequence of target machine instructions.

193. For the reasons described in §IX.C, Loderer discloses generating a sequence of target machine instructions to emulate condition codes on the target

machine and executing these instructions to compute the required values. *See* Ex. 1005, 1, Fig. 1, Fig. 4, col. 1 ¶3, col. 2 ¶4; col. 6 ¶¶2-5, col. 7 claim 1. As explained in §X.A, a POSA would have been motivated to extend Loderer, in view of PoO and Kane, to calculate condition codes for other IBM/390 instructions, such as the IBM/390 ADD instruction. The example instructions in §X.B illustrate how a POSA would detect overflow and calculate an overflow condition code. For the same reasons discussed in ground 1, the instructions disclosed in ground 2 would be executed by on the target hardware M2 in Loderer.

194. For example, Loderer Figures 1 and 4 disclose execution of the sequence of target machine instructions (program code PCZ) for execution on target hardware M2.

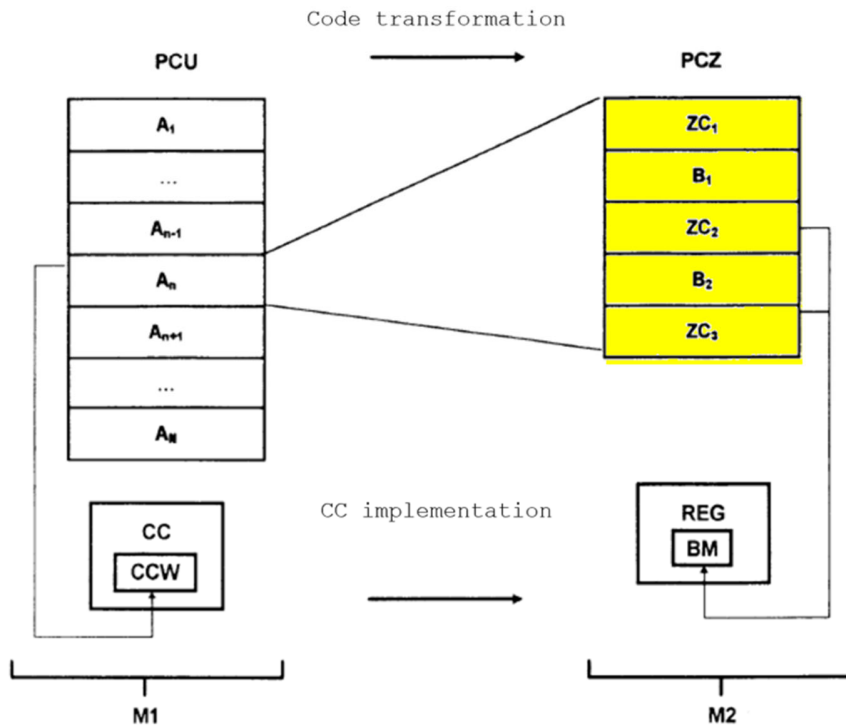


FIG. 1

Ex. 1005, Fig. 1.

195. “[A]ny instruction potentially influencing the value of the condition code is transformed by being simulated in the program code for the target hardware by means of a command generating a result value **in the course of the execution of the program** and without using branching commands.” *Id.*, col. 2 ¶1; *see also id.* col. 7 claim 1 (“each instruction (A_n) influencing the value (CCW) of the condition code (CC) in the program code (PCU) for the source hardware (M1) is simulated in the program code (PCZ) for the target hardware (M2) without using branching commands and using at least one command (B_1, B_2) **which generates a result value during the program execution**”), 1 (“generating a result value in the

course of the execution of the program”), col. 2 ¶4, col. 1 ¶3. The example instructions in §X.B are a modification of the target machine instructions disclosed in Loderer that would be executed on the target machine.

196. Thus, the combination of Loderer, PoO, Kane, and the knowledge of a POSA renders claim 4 obvious.

G. Claim 6: The computer program product of claim 1, wherein the sequence of target machine instructions together operate to directly calculate the target machine condition code using at least one of addition, subtraction, and multiplication.

197. As explained in §X.A and §X.B above, it would have been obvious to a POSA that an addition instruction is used to directly calculate the target machine condition code in the overflow case in calculating the condition code for a signed ADD.

Command sequence in PCZ		a = 0	a > 0	a < 0	Overflow
ZC ₁	Set BM to '0100'	BM = 0100	BM = 0100	BM = 0100	BM = 0100
B1	If a = 0, set t ₁ = 1, otherwise t ₁ = 0	t ₁ = 1	t ₁ = 0	t ₁ = 0	-
ZC ₂	BM << t ₁	BM = 1000	BM = 0100	BM = 0100	-
B2	If a > 0, set t ₂ = 1, otherwise t ₂ = 0	t ₂ = 0	t ₂ = 1	t ₂ = 0	-
ZC ₃	BM >> t ₂	BM = 1000	BM = 0010	BM = 0100	-
ZC ₂	BM << t ₃	BM = 1000	BM = 0010	BM = 0100	BM = xx00
ZC ₂	BM << t ₃	BM = 1000	BM = 0010	BM = 0100	BM = x000
ZC ₂	BM << t ₃	BM = 1000	BM = 0010	BM = 0100	BM = 0000
A _n	addu BM, BM, t ₃	BM = 1000	BM = 0010	BM = 0100	BM = 0001

198. In addition, as explained in §X.B, it would also have been obvious to a POSA to use addition in directly calculating the target machine condition code as detailed in the exemplary implementation to detect overflow and generate the overflow code t3 (stored in the form of associated bit pattern BM).

199. The combination of Loderer, PoO, Kane, and the knowledge of a POSA discloses or suggests every element of claim 6.

H. Claim 10: The computer program product of claim 1, wherein determining the intermediate condition code value comprises storing the intermediate condition code value in a condition code storage location as the provisional target machine condition code, and wherein determining the target machine condition code based on the distinguishing information and the intermediate condition code value comprises performing at least one operation on the intermediate condition code value stored in the condition code storage location to obtain the target machine condition code and storing the target machine condition code in the condition code storage location.

200. For the reasons described in ground 1 (§IX.E), Loderer discloses storing the intermediate condition code value (e.g., non-final version of BM) as a provisional target machine code in the condition code storage location (e.g., register BM) and performing an operation on the intermediate condition code value stored in the condition code storage location to obtain the target machine condition code and storing the target machine condition code in the condition code storage location. As explained in §X.A, a POSA would have been motivated to extend Loderer, in view of PoO and Kane, to calculate condition codes for other IBM/390 instructions, such as the IBM/390 ADD instruction. The example instructions in §X.B illustrate how a POSA would detect overflow and calculate an overflow code. For the same reasons discussed in ground 1, the instructions disclosed in §X.B meet the limitations of claim 10 because they store the intermediate condition code value (e.g., non-final version of BM) as a provisional target

machine code in the condition code storage location (e.g., register BM) and perform an operation on the intermediate condition code value stored in the condition code storage location to obtain the target machine condition code and storing the target machine condition code in the condition code storage location.

201. Accordingly, the combination of Loderer, PoO, Kane, and the knowledge of a POSA renders claim 10 obvious.

I. Claims 11, 16

202. Independent claims 11 and 16 contain similar limitations as claim 1. Accordingly, the combination of Loderer, PoO, and Kane renders obvious claims 11 and 16 for the same reasons described with respect to claim 1. *See supra* §IX.F.

11. A computer system for emulating computer instructions from a source machine to produce sequences of instructions on a target machine, the computer system comprising: a memory; and a processor in communication with the memory, wherein the computer system is capable of performing a method, the method comprising:	<i>See</i> paragraph above, §X.C (1[a1]-1[a2])
obtaining by the target machine a computer instruction from the source machine, said source	<i>See</i> §X.C (1[b])

machine having a different architecture from said target machine; and	
generating a sequence of target machine instructions which together operate to derive an encoding for a target machine condition code for the computer instruction, wherein the sequence of target machine instructions provides distinguishing information to distinguish between a plurality of possible outcomes for the target machine condition code, and directly calculates the target machine condition code without using branch instructions, the directly calculating comprising:	See §X.C (1[c]-[d])
determining an intermediate condition code value, the intermediate condition code value being a provisional value for the target machine condition code and subject to change based on the distinguishing information; and	See §X.C (1[e])
determining, based on the intermediate condition code value and based on the	See §X.C (1[f])

distinguishing information, the target machine condition code, wherein at least part of said distinguishing information is separate from the intermediate condition code value.	
---	--

J. Claims 12, 17

203. Claims 12 and 17 contain similar limitations as claim 2. Accordingly, the combination of Loderer, PoO, and Kane renders obvious claims 12 and 17 for the same reasons described with respect to claim 2. *See* §X.D.

XI. SECONDARY CONSIDERATIONS

204. It is my understanding that Patent Owner IBM hasn't produced or identified any evidence of secondary considerations of non-obviousness. Should the Patent Owner put forth any alleged evidence of secondary considerations in connection with the '289 patent, I reserve the right to consider such evidence, to consider whether any such evidence has a nexus to any claims of the '289 patent, and to update my analysis.

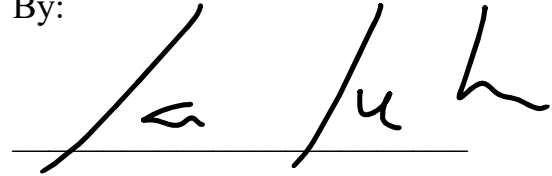
XII. CONCLUSION

205. All statements made herein of my own knowledge are true, and all statements made on information and belief are believed to be true. I further understand that willful false statements and the like are punishable by fine or imprisonment, or both under Section 1001 of Title 18 of the United States Code. I

IPR of U.S. Patent 8,713,289

declare under penalty of perjury that the foregoing is true and correct.

By:

A handwritten signature in black ink, appearing to be 'Ian Jestice', written over a horizontal line.

Ian Jestice

Date 11/28/2022