



UNITED STATES PATENT AND TRADEMARK OFFICE
BEFORE THE PATENT TRIAL AND APPEAL BOARD

LzLabs

v.

IBM

IPR2023-00274

U.S. Patent No. 8,713,289

Petitioner's Oral Argument – March 21, 2024

Two Asserted Grounds

Ground	Basis	References	Challenged Claims
1	§ 103	Loderer	1, 4, 6, 10-11, 16
2	§ 103	Loderer, PoO, Kane	1-4, 6, 10-12, 16-17

The Patent Owner Response (POR) raises a single dispute:

Whether Loderer discloses a “target machine condition code”

Proposed Agenda

- Overview of '289 Patent & Challenged Claims

- Loderer's "Target Machine Condition Code"

- IBM's Contentions

Emulating Condition Code Settings Without Branch Instructions

(12) **United States Patent**
Copeland et al.

(54) **EFFICIENTLY EMULATING COMPUTER
ARCHITECTURE CONDITION CODE
SETTINGS WITHOUT EXECUTING BRANCH
INSTRUCTIONS**

(75) Inventors: **Reid T. Copeland**, Toronto (CA);
Patrick R. Doyle, Toronto (CA);
Charles B. Hall, Calgary (CA); **Andrew
Johnson**, Hampshire (GB); **Ali I.
Sheikh**, Toronto (CA)

(73) Assignee: **International Business Machines
Corporation**, Armonk, NY (US)

(*) Notice: Subject to any disclaimer, the term of this
patent is extended or adjusted under 35
U.S.C. 154(b) by 1187 days.

(21) Appl. No.: **11/668,623**

(22) Filed: **Jan. 30, 2007**

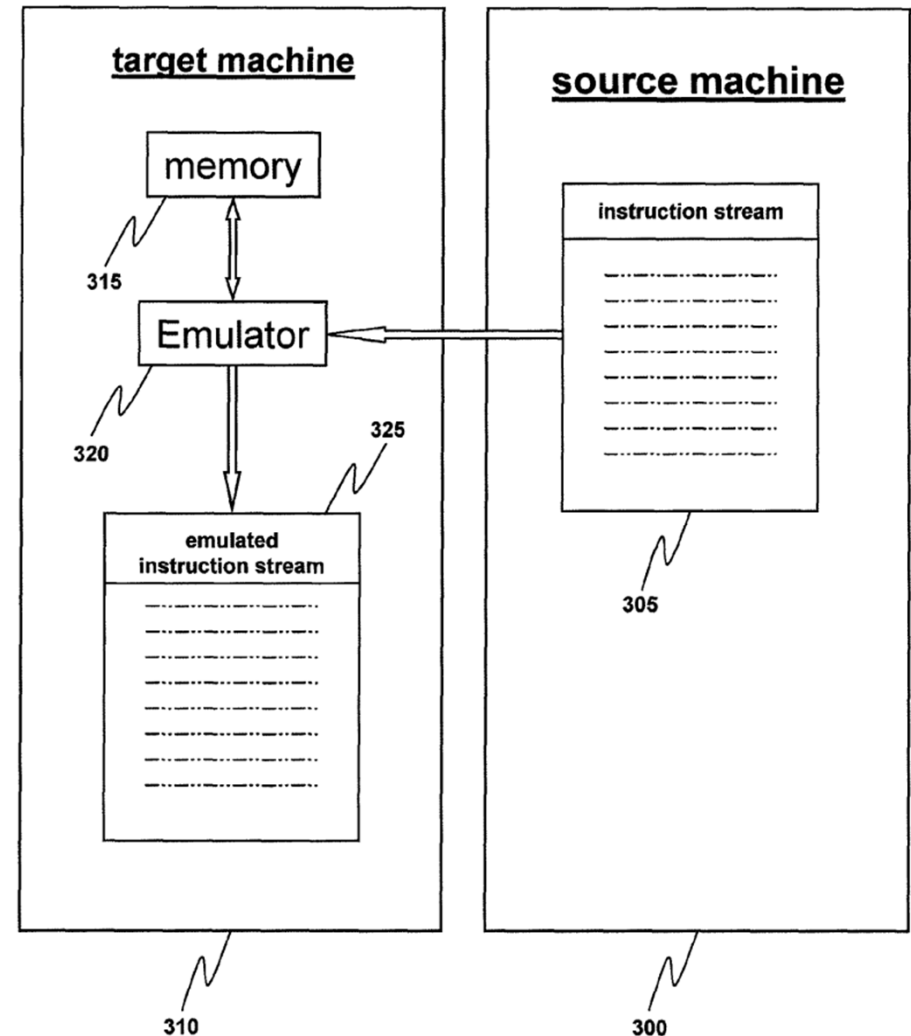
tions written for another system. The present invention relates
to the handling of condition code settings in the context of
instruction emulation. While the systems and methods of the
present invention are widely applicable to any emulation 45
method where condition codes are present, it is particularly
applicable to the emulation of the z/Architecture. However,
the principles set forth herein are applicable to any source
architecture and to any target architecture.

‘289 Patent, 1:42-49; Petition, 7-9; Reply, 3, 5, 27

Emulation

The typical emulation environment in which the present invention is employed is illustrated in FIG. 5. Emulators such as 320 accept, as input, instruction streams 305 representing machine or assembly language instructions which are designed to operate on source machine 300. Emulator 320 employs memory 315 in target machine 310 to produce a stream of instructions which are capable of executing on target machine 310. While FIG. 5 particularly shows opera-

'289 Patent, 8:64-9:4, Fig. 5



Emulating Condition Code Settings Without Branch Instructions

Accordingly, it is seen that the designer of emulation systems is faced with the paradoxical choice of needing branch
35 instructions to successfully emulate the generation of condition code settings in target architectures while at the same time desiring to avoid branching instructions because of their disadvantages. This problem is especially severe when condition code generation and functionality in the target architecture
40 are quite different from that found in the architecture of the source machine.

'289 Patent, 2:33-41; Reply, 4, 8

Why Emulate Condition Code Settings on the Target Machine?

When emulating instructions from a source machine ISA on a target machine ISA system whose condition codes are different from those of the source machine ISA, it is necessary to have a target machine representation (encoding) of the source machine ISA condition code. This representation of the source machine ISA condition code is necessary to achieve accurate emulation on the target machine of source machine ISA instructions that depend on the specific settings and meanings of the source machine ISA condition code. It is this representation of the source machine ISA condition code that is referred to here as the “target machine condition code.”

EX1028 (IBM Infringement Contentions), 4-5; Reply, 24-25

Emulating Condition Code Settings in the '289 Patent

A classic example of an instruction which produces condition code changes upon execution is the compare instruction which sets a condition code to “zero” if the operands are equal, to “one” if the first operand is strictly less than the second operand and to “two” if the first operand is strictly greater than the second operand. 30

'289 Patent, 1:29-34

COMPARE LOGICAL

Resulting Condition Code:

0	Operands equal
1	First operand low
2	First operand high
3	--

EX1026, 78-79; Reply, 9, 12

Emulating Condition Code Settings in the '289 Patent

dition codes. Accordingly, a sequence for emulating condition code settings is provided below.

COMPARE LOGICAL

Resulting Condition Code:

- 0 Operands equal
- 1 First operand low
- 2 First operand high
- 3 --

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

“1” and placing the result “0” in rC. Thus, at the end of the instruction sequence set out above, the following results are obtained: rC=2 if rA>rB; rC=1 if rA<rB; and rC=0 if rA=rB. 10

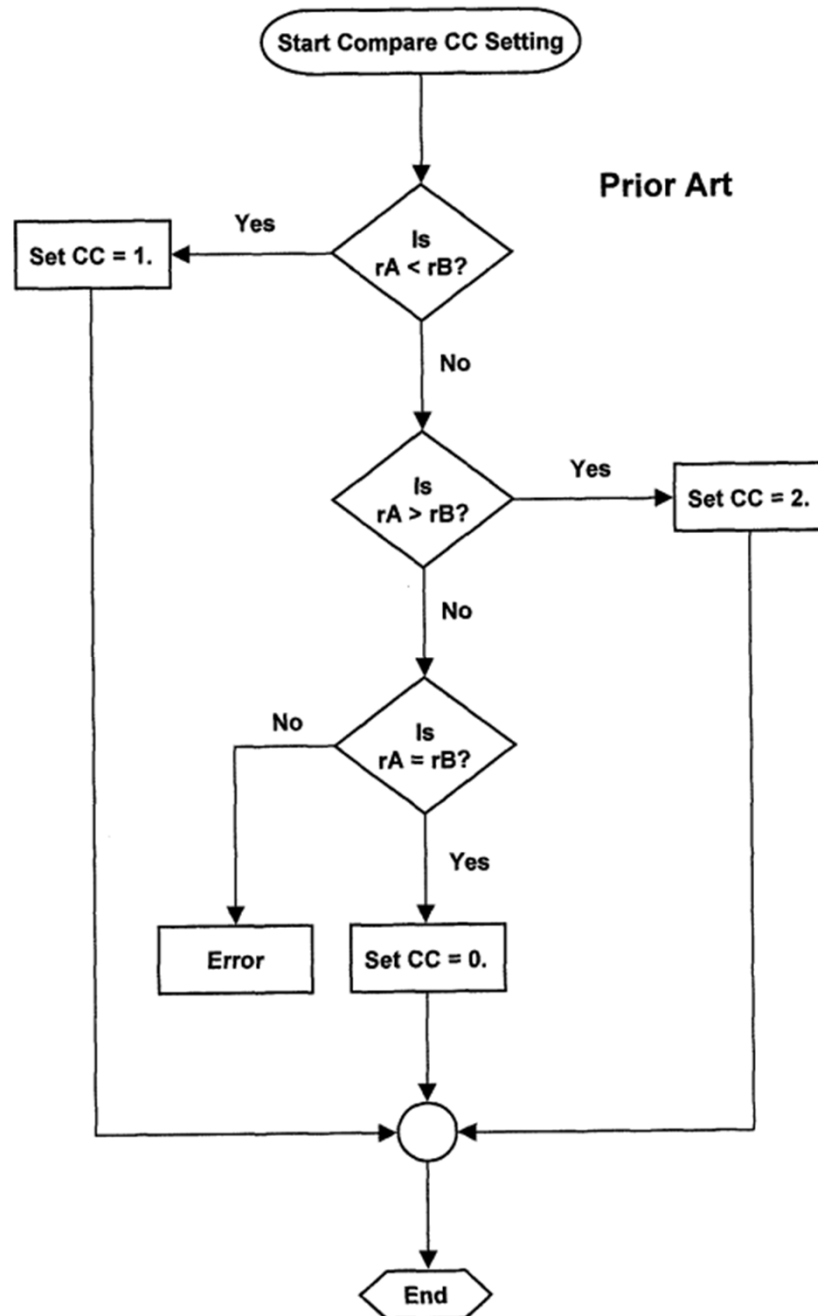
EX1026, 78-79; '289 Patent, 5:32-55, 7:8-10; Petition, 5-7, 9, 31-33, 37; Reply, 4, 5, 8-10, 12

Condition Codes Are Stored As Bits of Data

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 20
the generation of several bits of data to indicate the outcome
status of instruction execution. These bits are typically
referred to as condition codes.

‘289 Patent, 1:18-23, Petition, 4; Reply, 12, 23, 27

Branch Instructions



'289 Patent, Fig. 1

Emulating Condition Code Settings Without Branch Instructions

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:

20

a non-transitory computer readable storage medium readable by a processor and storing instructions for execution by the processor for performing a method comprising:

obtaining by the target machine a computer instruction

25

from the source machine, said source 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

30

instruction, wherein the sequence of target machine instructions provides distinguishing information



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

35

using 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

40

and subject to change based on the distinguishing information; and



Intermediate
condition code value

determining, based on the intermediate condition code value and based on the distinguishing information, the target machine condition code, wherein

45

at least part of said distinguishing information is separate from the intermediate condition code value.



Target machine
condition code

'289 Patent, Claim 1;
Petition, 22-38; Reply 5-6

Emulating Condition Code Settings Without Branch Instructions

Petition, 22-38; Reply 5-7

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:

20

a non-transitory computer readable storage medium readable by a processor and storing instructions for execution by the processor for performing a method comprising:

obtaining by the target machine a computer instruction

25

from the source machine, said source 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

30

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

35

using branch instructions, the directly calculating

comprising:

determining an intermediate condition code value, the

intermediate condition code value being a provi-

40

sional value for the target machine condition code

and subject to change based on the distinguishing

information; and

determining, based on the intermediate condition

code value and based on the distinguishing infor-

45

mation, the target machine condition code, wherein

at least part of said distinguishing information is

separate from the intermediate condition code

value.

Directly calculates the target machine condition code...by:

(1) Determining an intermediate condition code value

(2) Determining the target machine condition code based on the intermediate code and distinguishing information

Disclosed Embodiment of the Challenged Claims

code. To illustrate, refer to the tables presented in paragraphs [0025] and [0031] in applicants' specification. The target machine condition code, as a result of the sequence of instructions, will be held in register rC. An intermediate condition code value for that target machine condition code is determined (e.g. by performing the first four instructions in these examples) and placed in rC. That intermediate condition code value is a provisional value for the target machine condition code – that is, it is not the target machine condition code, *per se*, but rather a provisional value for the target machine condition code because it is subject to change based on the distinguishing information (the value stored in register rX in these examples). Based on performing the final instruction, in these examples, the target machine condition code is determined and stored in rC, which may or may not be a modified version of the intermediate condition code value that was determined and stored in rC prior to performance of the final instruction.

rC after
instruction [4]

Value stored
in rX

rC after
instruction [5]

EX1004, 83-84; Petition 31; Reply, 11, 13, 28

Disclosed Embodiment of the Challenged Claims

Distinguishing
information in rX

Intermediate
cc value in rC

Target machine cc in
rC = emulated IBM
mainframe cc
generated by taking
difference of rX and
rC—[5] sub rC, rC, rX

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

COMPARE LOGICAL

Resulting Condition Code:

- 0 Operands equal
- 1 First operand low
- 2 First operand high
- 3 --

'289 patent, 5:41-56;
EX1026, 78-79; Petition,
5-7; Reply 8-9, 12

Undisputed: The target machine condition code is held in register rC at the end of the instruction sequence

To illustrate, refer to the tables presented in paragraphs [0025] and [0031] in applicants' specification. The target machine condition code, as a result of the sequence of instructions, will be held in register rC.

EX1004, 83; Petition, 31; Reply, 11, 13, 28

“1” and placing the result “0” in rC. Thus, at the end of the instruction sequence set out above, the following results are obtained: rC=2 if rA>rB; rC=1 if rA<rB; and rC=0 if rA=rB. 10

'289 Patent, 7:8-10; Petition, 7, 31-33; Reply, 4, 5, 8, 10, 12

Undisputed: The target machine condition code is a representation of the IBM mainframe (source machine architecture) condition code

In particular, the subject source instruction is the z/Architecture Instruction called the Compare Logical operation. ³⁵ *The example assumes that rA and rB are zero-extended*

'289 Patent, 5:34-36

COMPARE LOGICAL
Resulting Condition Code:

0	Operands equal
1	First operand low
2	First operand high
3	--

Instruction	<u>rA > rB</u> Register Contents			<u>rA = rB</u> Register Contents			<u>rA < rB</u> 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

EX1026, 78-79

'289 Patent, 5:40-56

Petition 5, 6, 9, 32, 37; Reply, 4, 8-10, 12

Target Machine Condition Code = representation of the source machine architecture condition code

When emulating instructions from a source machine ISA on a target machine ISA system whose condition codes are different from those of the source machine ISA, it is necessary to have a target machine representation (encoding) of the source machine ISA condition code. This representation of the source machine ISA condition code is necessary to achieve accurate emulation on the target machine of source machine ISA instructions that depend on the specific settings and meanings of the source machine ISA condition code. It is this representation of the source machine ISA condition code that is referred to here as the “target machine condition code.”

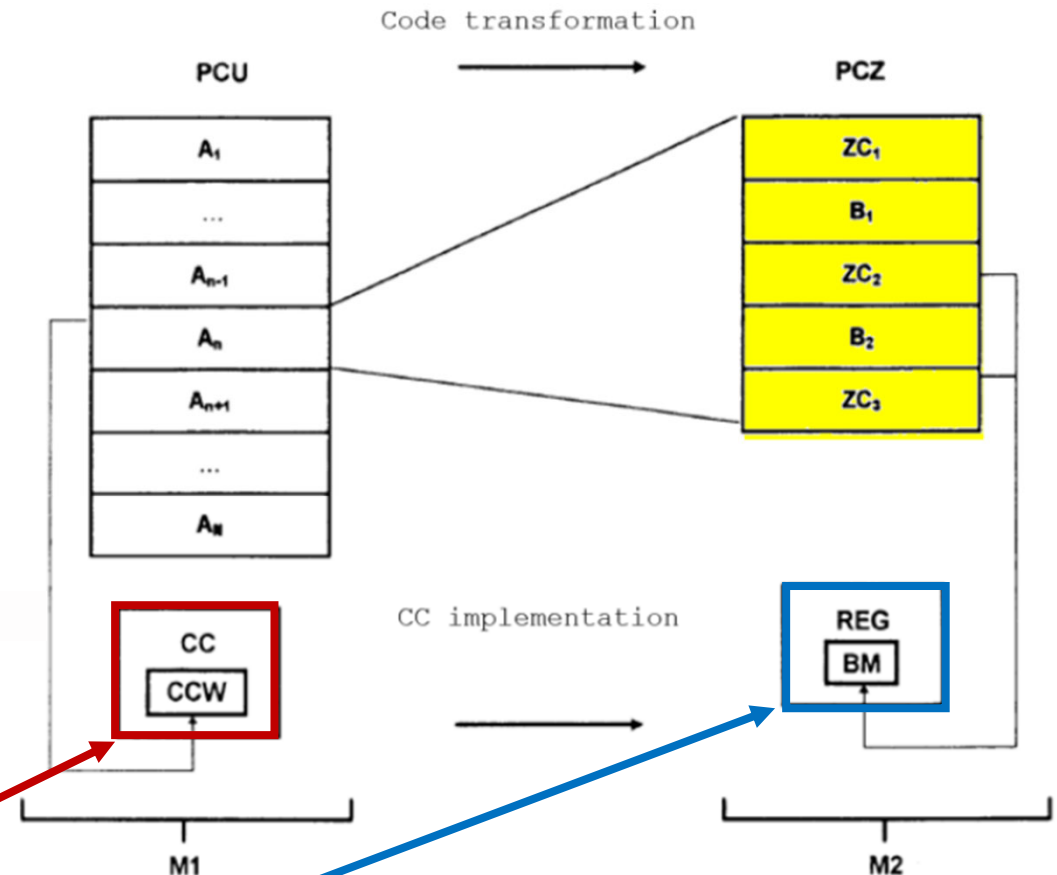
EX1028 (IBM Infringement Contentions), 4-5;
Reply 24-25

Loderer's Target Machine Condition Code

Loderer Solved the Same Problem 10 Years Earlier

Loderer's target machine **M2** generates an encoding of the emulated IBM mainframe **M1** condition code:

- (1) using a sequence of instructions; and
- (2) without branch instructions



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

FIG. 1

FIG. 2

EX1005, Figs. 1, 2;
Petition, 12, 18, 23, 26,
28, 30; Reply, 14-15

Loderer Eliminates Branch Instructions

When porting the program code (PCU), any instruction (A_n) potentially influencing the value (CCW) is transformed by being simulated in the program code (PCZ) by means of a command (B_1 , B_2) generating a result value in the course of the execution of the program and without using branching commands.

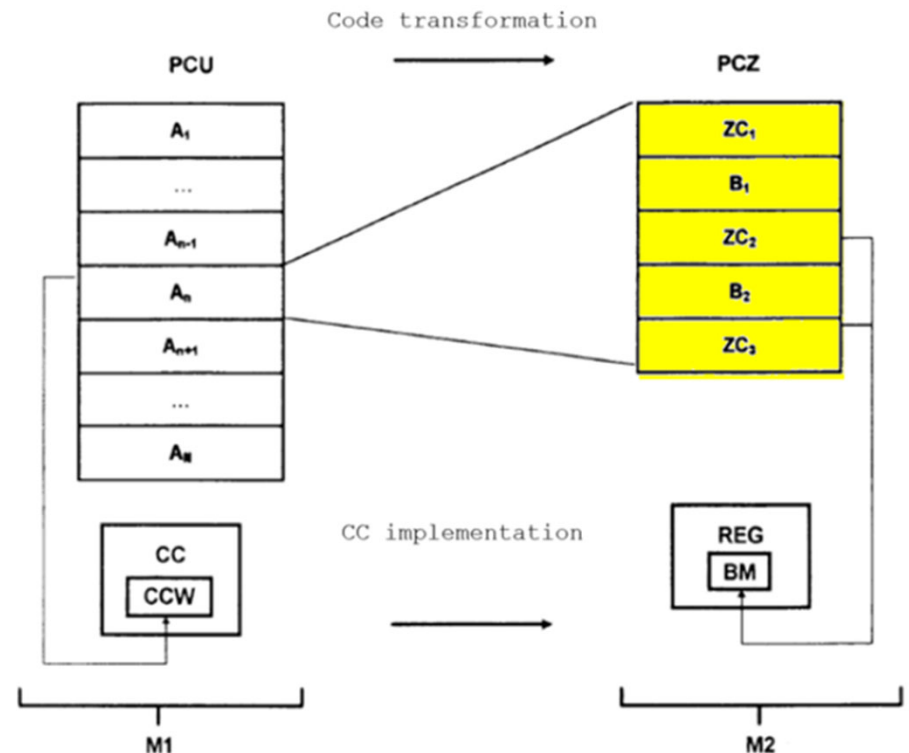


FIG. 1

EX1005, Fig. 1; Petition, 12, 18, 23, 26, 28, 30, 39, 40, 50; Reply, 14-15

Loderer's Register BM holds a representation of the IBM mainframe (source machine architecture) condition code

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).

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

FIG. 2

EX1005, 1, Fig. 2; Petition, 10, 18, 23-25, 29, 30, 33, 40, 51, 56, 73

Loderer Discloses Target Machine Condition Codes

These 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.

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; Ex. 1003, ¶102. “Mapping is defined which uniquely

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

FIG. 2

Petition, 29; EX1005, Fig. 2

Loderer Discloses Target Machine Condition Codes

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.

EX1003, ¶98

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”

EX1003, ¶76

EX1003, ¶102

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

Loderer Discloses Target Machine Condition Codes

Undisputed: At the end of the PCZ sequence on the target machine, Loderer's register BM contains bits representing the emulated IBM mainframe condition code.

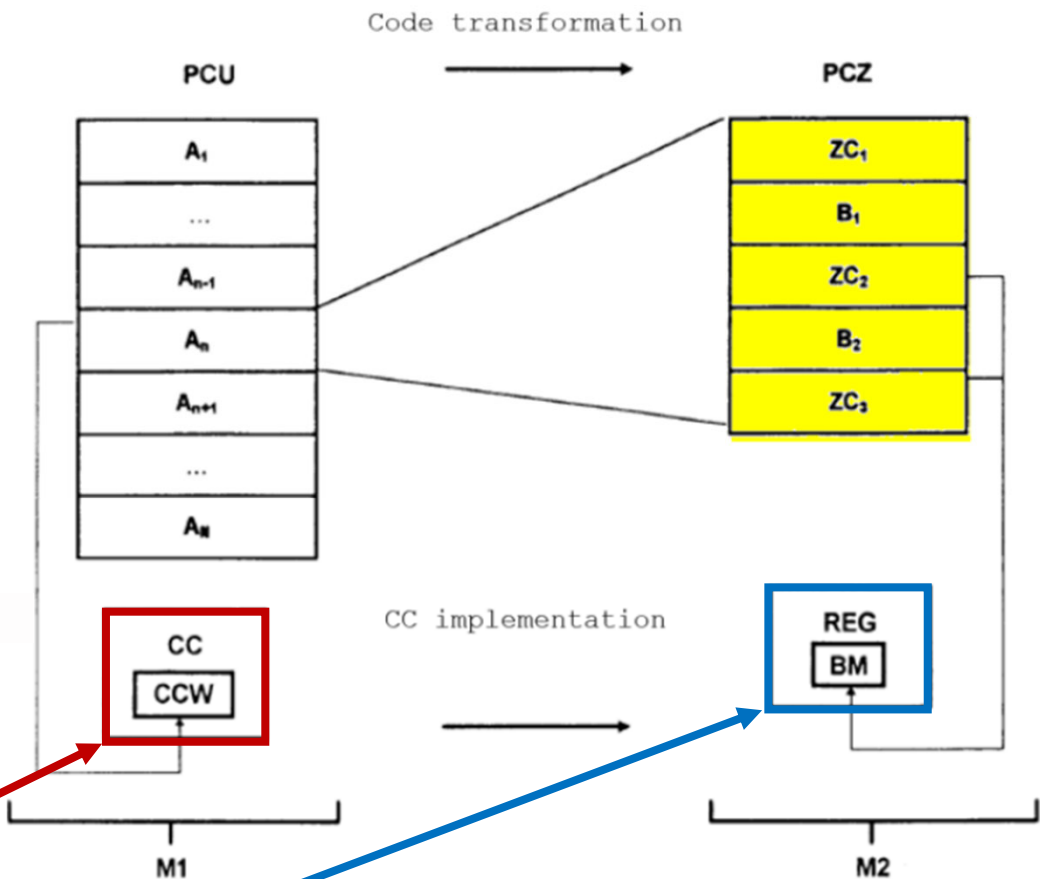


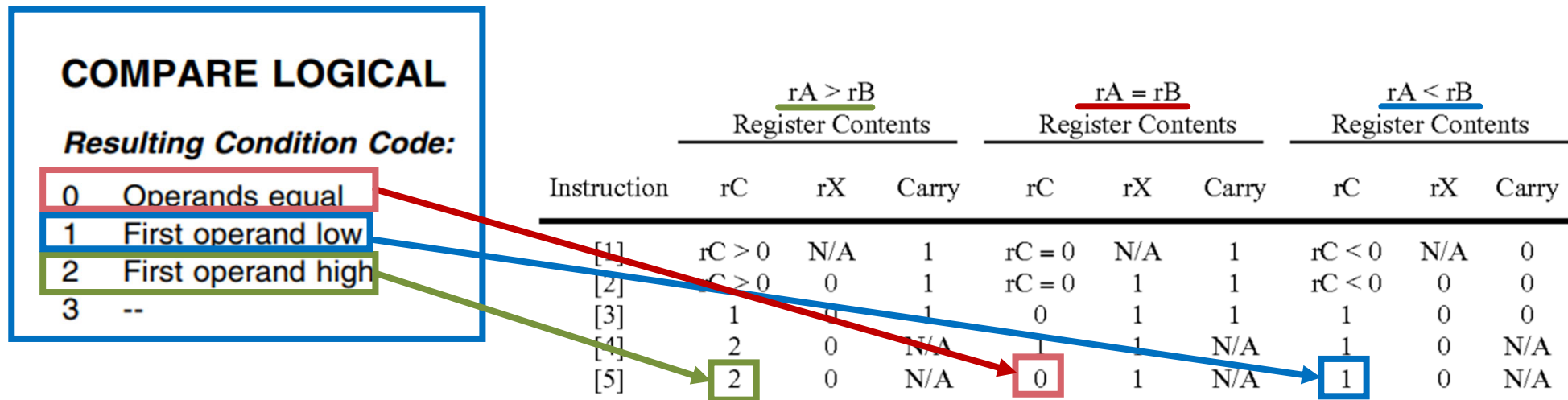
FIG. 1

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

FIG. 2

EX1005, Figs. 1, 2;
Petition, 12, 18, 23, 26,
28, 30; Reply, 14, 15

Loderer's Target Machine Condition Codes Are Just Like the Disclosed Embodiment of the Challenged Claims



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

FIG. 2

EX1026, 78-79; '289 patent, 5:40-56; EX1005, Fig. 2; Petition 5-7, 9, 32, 37; Reply, 4, 8-10, 12

The Challenged Claims Do Not Limit How the Target Machine Condition Code is Encoded

generating a sequence of target machine instructions which together operate to derive an encoding for a target machine condition code for the computer 30 instruction, wherein the sequence of target machine

'289 patent, 9:28-31

in their examples. The examples provided in the specification for the present invention describe how to derive an encoding of 0, 1 or 2 (with binary bit patterns of 00, 01, 10, respectively) given only the relative ordering of two operands in the emulated instruction.

EX1004, 212; Reply, 11

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

EX1005, Figure 2

Claim Construction is Unnecessary

1. The challenged claims map directly onto Loderer's sequence of instructions.
2. Loderer derives an encoding for emulated IBM mainframe condition codes, just like the '289 patent.
3. The claim defines **how** to directly calculate the target machine condition code and does not limit **where** it is stored.

'289 Patent, Claim 1;
Petition, 18-38; Reply, 5-7, 13-17

Petitioner's Contention

Both parties contend “target machine condition code” should be given its plain and ordinary meaning, which requires reading the term in the context of the specification and prosecution history. *Thorner v. Sony Computer Entm’t Am. LLC*, 669 F.3d 1362, 1365 (Fed. Cir. 2012). But the POR and PO’s expert do not address the term in the context of the preferred embodiments or address the portions of the intrinsic record that undercut PO’s position, instead relying on a table from an extrinsic textbook to distinguish Loderer. To the extent the Board determines a construction is necessary to resolve the parties’ dispute, the Board should interpret “target machine condition code” consistent with the intrinsic record and PO’s litigation position to include bits of data stored on the target machine that indicate the outcome status of executing the source machine instruction on the source machine (i.e., “an IBM mainframe condition code representation on the target machine”). The Board should reject PO’s attempt to limit the target machine architecture and/or the target machine condition code storage location based on

Reply, 26

IBM Misrepresents Petitioner's Contention

In the Reply, Petitioner advances a new claim construction position for the term “target machine *condition code*” as “encoded values, stored as bits in a register, corresponding to *condition codes* that would have been generated from executing instructions on the source machine.” Reply, 1.

Sur-Reply, 3

Patent Owner's Response (“POR”) raises a single dispute: whether Loderer discloses a “target machine condition code” of the challenged claims. As the petition explains, Loderer's target machine condition codes are just like those of the '289 patent—*i.e.*, encoded values, stored as bits in a register, corresponding to condition codes that would have been generated from executing instructions on the source machine. Patent Owner's (“PO”) position in this IPR is directly contrary to PO's

Reply, 1

IBM's Contentions

IBM's Contentions – Patent Owner Response

- **No IBM Preliminary Response**

- **IBM's POR and expert:**

1. Neither discusses the preferred embodiment or register rC, and neither disputes the Petition's mapping of the Table II embodiment to the challenged claims.
2. Neither disputes that Loderer's target machine generates a sequence of instructions that derive an encoding for an emulated IBM mainframe condition code without executing branch instructions.
3. Both rely solely on extrinsic evidence to conclude "condition codes" are "special bits," but neither explains how the bits stored in rC are more "special" than the bits in Loderer's register BM.
4. In the end, the POR merely "proves" the undisputed proposition that the MIPS processor of Loderer's target machine does not have dedicated hardware support for condition codes (e.g., a "condition code register").

POR, 2-5, 18, 25, 30; EX2001, 13-15, 28, 34; Reply, 13-17, 20, 26

IBM wrongly equates “target machine condition code” with “target architecture condition code”

Because Loderer lacks a target architecture condition code, Loderer teaches a “method for simulating a condition code.” EX1005, 2. Specifically, Loderer

POR, 19

That is, independent claim 1 relates to emulating program code from a source architecture that supports condition codes to a target architecture that also supports condition codes to specify the branch conditions, and explicitly requires “directly calculat[ing] the target machine condition code.” EX1001, 9:34-35. EX2001, ¶44.

POR, 11

Target Machine

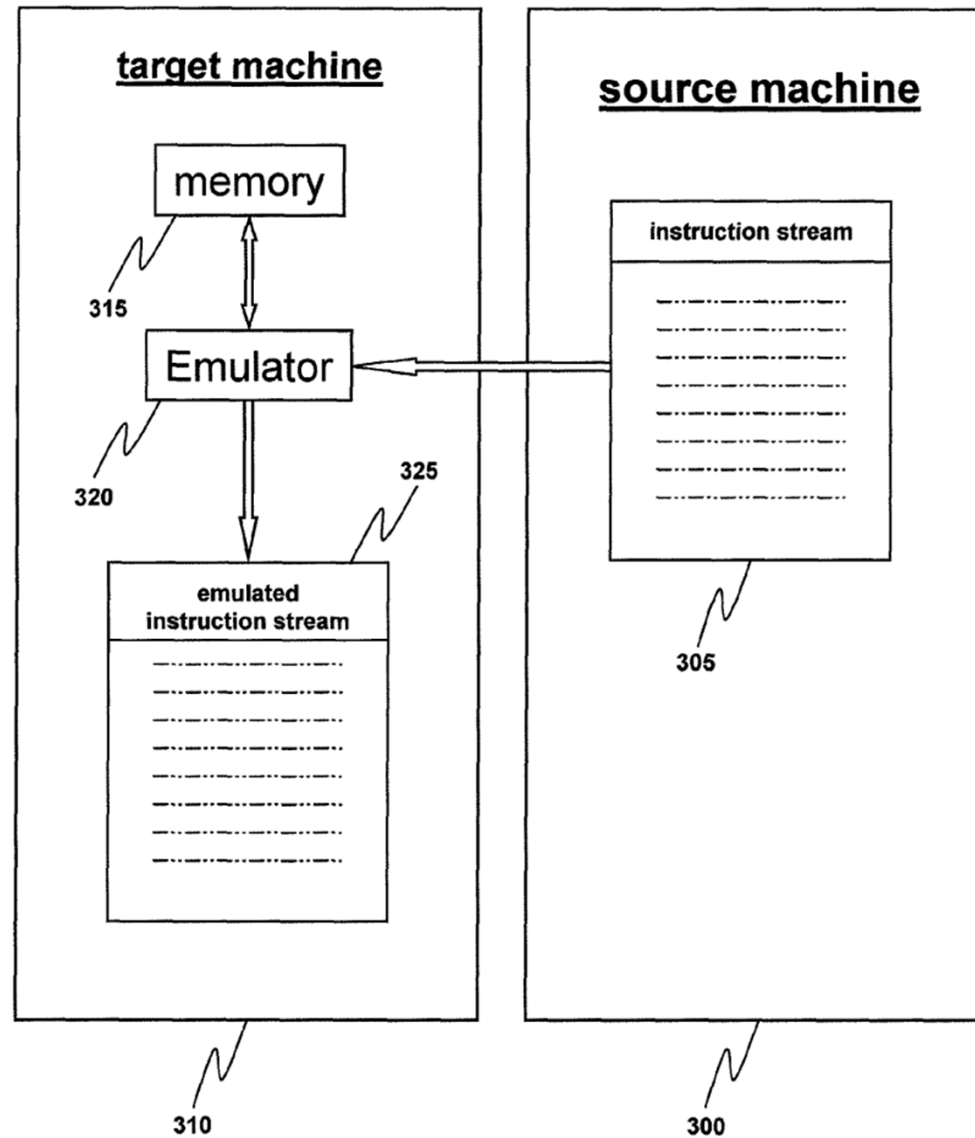


Fig. 5

'289 patent, Fig. 5

Target Machine Condition Code is an Emulated IBM Mainframe Condition Code Encoded as Bits

dition codes. Accordingly, a sequence for emulating condition code settings is provided below. The sequence provided

COMPARE LOGICAL

Resulting Condition Code:

- 0 Operands equal
- 1 First operand low
- 2 First operand high
- 3 --

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

in their examples. The examples provided in the specification for the present invention describe how to derive an encoding of 0, 1 or 2 (with binary bit patterns of 00, 01, 10, respectively) given only the relative ordering of two operands in the emulated instruction.

‘289 Patent, 5:34-36, 40-56; EX1026, 78-79; EX1004, 212;
Petition 5, 6, 9, 32, 37; Reply, 4, 8-12

IBM Contends “Target Machine Condition Code” Requires Specialized Hardware

source machine. Despite sidestepping claim construction entirely in the POR⁶, PO apparently interprets “target machine condition code” as limiting the architecture of the target machine to an architecture that provides certain dedicated support for condition codes, such as a program-status word (“PSW”) in the IBM/390 or a dedicated “condition code register” in other systems. See POR, 5-6, 16-23.

According to PO, a condition code value that is directly calculated without branch instructions but stored in a “condition register” or general-purpose register at the target machine cannot, by definition, be a “target machine condition code” of the challenged claims. See POR, 5-6 (PO arguing that MIPS uses only “condition

Reply, 18

IBM Contends “Target Machine Condition Code” Requires Specialized Hardware

....” Petition, 18. The MIPS architecture provides no condition codes. *See* EX1009, 43 (“The R2000² architecture also provides no condition code register containing status bits such as Carry and Overflow.”)

EX2001 (Goodin Decl.), ¶59

But The '289 Patent Expressly States Condition Code Registers Are Not Required

35 In some cases it is possible and efficient to manipulate the
PowerPC condition code register itself to derive the z/Archi-
tecture condition code settings. In these cases a PowerPC
record form instruction is used and the resulting PowerPC
condition register is manipulated by rotations and logical
40 tion code setting.

'289 Patent, 4:33-40; Reply, 10

and machine states. It is noted that while an emulation envi-
ronment preferably results in the setting of hardware or con-
dition code elements in the target architecture, the present
invention also contemplates the situation in which condition 60
codes are generated and stored in locations other than condi-
tion code registers in the target machine.

'289 Patent, 1:57-62; Reply, 10, 12, 28

The MIPS Processor Does Not Provide Dedicated Hardware Support for Condition Codes

LzLabs does not dispute that both parties' experts agree Loderer discloses a ~~target machine~~ that does not provide condition codes. See EX2003, 89:4-7;
EX2001, ¶57; POR, 35. LzLabs also does not dispute that the Petition maps the

Sur-reply, 2

57. Therefore, Loderer ports program code from a source architecture that supports condition codes to a target architecture “in which *no hardware support of a condition code is provided.*” EX1005, 2. As Mr. Jestice, LzLabs' expert, testified, Loderer discloses a target architecture with no condition codes:

Q. ... The invention described in Loderer relates to *a target hardware that is not providing a condition code, correct?*

A. *That's correct.*

EX2003, 89:4-7.

EX2001, ¶57;
POR, 17

The Challenged Claims Are Not Limited to Any Particular Target Architecture

ture. These techniques are specifically directed to situations
in which the PowerPC architecture and the Intel IA32 archi- 25
tectures are employed to emulate the z/Architecture. The
sequences of the present inventions are more efficient and
generally smaller as opposed to a more straightforward
method that requires more flow control. However, it is noted
that the principals, techniques and methods of the present 30
invention are not limited to any particular target machine
architecture. The two exemplar architectures discussed

‘289 patent, 3:24-32; Petition, 5; Reply, 5, 9, 10, 27

The Challenged Claims Are Not Limited to Any Particular Target Architecture

instruction emulation. While the systems and methods of the present invention are widely applicable to any emulation 45 method where condition codes are present, it is particularly applicable to the emulation of the z/Architecture. However, the principles set forth herein are applicable to any source architecture and to any target architecture.

‘289 patent, 1:44-49; Reply, 5, 27

Accordingly, the last sentence, which allegedly forms the basis for LzLabs’ argument, merely discloses that the invention principles are not limited to the emulation of the z/Architecture but are applicable to other target architectures “where condition codes are present.”

Sur-reply, 11

IBM Pivots in its Sur-Reply

IBM's Sur-Reply does not:

1. Mention “special bits” like the POR.
2. Dispute the target machine condition code in rC is a representation of an IBM mainframe condition code.
3. Dispute the bit pattern in Loderer's BM is a representation of an IBM mainframe condition code.

IBM's Sur-Reply does:

1. Raise new arguments about generating condition codes by “setting CPU flag bits” in the target processor.
2. Contend a target machine condition code cannot be a bit pattern in a general-purpose register.

IBM Pivots to “Generating Condition Codes”

misapprehension of IBM’s arguments. Specifically, LzLabs characterizes IBM’s argument as contingent on storing condition codes in a “special type of storage.” Reply, 26. To the contrary, as explained below, IBM’s position is that the target machine provides a “condition code;” not whether condition code values are, after being generated, stored in a particular location.

Sur-reply, 1-2

LzLabs argues that the “[t]arget machine condition code” of the ’289 patent thus includes emulated condition code settings on the target machine that would have resulted from execution of the source machine instruction on the source machine.” Reply, 9. That argument is a red herring. That the “target machine condition code” is an emulated condition code has no bearing on whether the target machine condition code is generated or not.

Sur-reply, 12-13

IBM Pivots to “Generating Condition Codes”

dition code elements in the target architecture, the present invention also contemplates the situation in which condition codes are generated and stored in locations other than condition code registers in the target machine. 60

‘289 patent, 1:57-62; Reply, 10, 12, 28

Petitioner’s quote expressly states “*condition codes are generated* and stored in locations other than condition code registers in the target machine.” That is, *after the condition codes are generated in the target machine*, e.g., in CPU flag bits or in a condition code register, they can be stored in other registers, e.g., maybe for accessing them or manipulating them from a more convenient location. Specifically, the ‘289 Patent discloses that “*an*

Sur-reply, 14

The Challenged Claims Do Not Require “Generating” Condition Codes

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:

20

a non-transitory computer readable storage medium readable by a processor and storing instructions for execution by the processor for performing a method comprising:

obtaining by the target machine a computer instruction 25

from the source machine, said source 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 30

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: 35

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 40

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. 45

‘289 Patent, Claim 1;
Petition, 6-7, 22-38; Reply, 5-7

Distinguishing
information

Intermediate
condition code value

Target machine
condition code

The Challenged Claims Do Not Require “Target CPU flag bits”

place. Consistent with the claims, which require a “target machine condition code” and with the Specification, which discloses that “an instruction ... in the target machine architecture ... sets one or more target CPU flag bits,” IBM’s argument is that Loderer does not satisfy the claimed “target machine condition code,” because it discloses a target machine that does not provide condition codes. POR, 26-34.

Sur-reply, 10

The Challenged Claims Do Not Require “Target CPU flag bits”

9. The computer program product of claim 8, wherein said first set of one or more possible outcomes comprises one case result and said second set of one or more possible outcomes comprises at least two other case results, and wherein said generating comprises:

mimicking an instruction present in a source instruction stream in a manner which sets one or more target central processing unit (CPU) flag bits and which stores a first result in a first storage location;

‘289 patent, Claim 9 (unchallenged); Reply, 12, 19

IBM's Contention: Target Machine Condition Code Cannot be a Bit Pattern in a General-Purpose Register

machine, as PO contends.” Reply, 7. IBM’s argument is that the plain and ordinary meaning of “target machine condition code” is not a bit pattern in a general-purpose register, which is consistent with Mr. Jestice’s testimony that a bit pattern is not a condition code. *See* EX2003, 86:5-87:2.

Sur-reply, 12

Why IBM is Wrong: Bit Pattern

11 Q. How is the condition code on the
12 source hardware different from the bit pattern
13 on the target hardware?

14 A. The bit pattern on the target
15 hardware has a single bit for each condition.
16 The condition code on the source architecture,
17 if it were run, has a condition code. Loderer
18 is describing it in coding the bit pattern, one
19 bit for each condition, into a condition code
20 that the target architecture -- or the source
21 architecture would have recognized.

22 Q. So you're saying because the bit
23 pattern in the target architecture uses a single
24 bit for its condition, Loderer chooses to call
25 it a bit pattern instead of a condition code?

1 A. It isn't a condition code. It's a
2 bit pattern.

EX2003, 86:11-87:2

Why IBM is Wrong: Bit Pattern

12 Q. Well, is the target machine condition
13 code BM?

14 A. It's a representation of it, yes.

15 Q. What do you mean a representation?

16 A. The target machine condition code
17 will end up in BM, so that is the target
18 condition code.

EX2003, 99:12-18

Why IBM is Wrong: Bit Pattern

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

EX1003, ¶102; Institution Decision, 15-17

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

EX1005, Figure 2

Why IBM is Wrong: Bit Pattern

in their examples. The examples provided in the specification for the present invention describe how to derive an encoding of 0, 1 or 2 (with binary bit patterns of 00, 01, 10, respectively) given only the relative ordering of two operands in the emulated instruction. This is to be contrasted with the patent to Liu et al., where individual condition code bits are simply transferred or trivially derived from similar incoming intermediate flags. In this regard, Applicant's paragraph [0031] is particularly worthy of consideration and comparison.

EX1004, 212; Reply, 11, 23

Why IBM is Wrong: General-Purpose Register

of a condition code setting.” *Id.*, 4:22-25. The disclosed technique of the ’289 patent results in generating and storing an encoding of the condition code setting in register rC, a general-purpose register on the target machine:

rA, rB—the first and second operand, respectively, of the z/Architecture instruction evaluated into a register;
rT—the result value of the z/Architecture instruction computed into a register;
rX, rY, rZ—temporary registers used to hold intermediate results;
rC—the register that will hold the condition code value at the end of the sequence.

Id., 5:20-29. Indeed, register rC of the Table II embodiment stores the result of the subtraction of instruction [1], rC is an operand in instructions [4] and [5], and rC holds the condition code value at the end of the sequence. *Id.*, 6:23-25, 6:59-62, 7:3-8.

Reply, 4-5

Why IBM is Wrong: General-Purpose Register

to such embodiments. *See* EX1001, 1:57-62, 3:29-34. Indeed, in the embodiment described above, the target machine condition code is stored in general-purpose register rC, which is the same register used to store the result of an arithmetic operation in instruction [1] and as an operand in instructions [4] and [5]. *See* EX1001, 5:39-56, 6:23-25, 6:59-62, 7:3-10, 7:25-49. Moreover, the '289 patent

Reply, 10

Why IBM is Wrong: General-Purpose Register

LzLabs argues that “[t]he disclosed technique of the ’289 patent results in generating and storing an encoding of the condition code setting in register *rC*, ***a general-purpose register on the target machine.***” Reply, 4. LzLabs is mistaken. LzLabs’ made-up attorney argument is not supported by the evidence. Tellingly, the Reply does not point to the Specification to support LzLabs’ contention that *rC* is a “general-purpose register.” There is no such disclosure. Instead, the Specification discloses that an instruction executed in the target machine ***sets one or more target CPU flag bits*** and the result is placed in register *rC*. Accordingly, register *rC* holds the result of one or more target CPU flag bits. EX1001, 7:51-55

Sur-reply, 8-9

Why IBM is Wrong: General-Purpose Register

rA, rB—the first and second operand, respectively, of the z/Architecture instruction evaluated into a register;
rT—the result value of the z/Architecture instruction computed into a register;
rX, rY, rZ—temporary registers used to hold intermediate results;
rC—the register that will hold the condition code value at the end of the sequence.

'289 Patent, 5:20-28; Petition, 37; Reply, 4, 8, 28

Why IBM is Wrong: General-Purpose Register

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

With respect to instruction [1], its execution sets the carry bit (that is, the CPU carry bit) to “1” in the case that $rA \geq rB$ and to “0” in the case that $rA < rB$. Additionally, rC contains the result of the subtraction, which, notably could be “0.” The

‘289 Patent, 5:39-56, 6:22-26; Petition, 6, 9, 32, 37;
Reply, 4, 8, 9, 10, 28

Why IBM is Wrong: General-Purpose Register

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 immmed	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

The execution of instruction [5] (sub), with operands “rC, rC, rX” provides the last step in which the contents of rX, now denoting equality, are subtracted from rC as mechanism for distinguishing the case that rA=rB from the case that rA<rB since the case of equality results in the subtraction of “1” from “1” and placing the result “0” in rC. Thus, at the end of the instruction sequence set out above, the following results are obtained: rC=2 if rA>rB; rC=1 if rA<rB; and rC=0 if rA=rB.

‘289 Patent, 5:39-56, 7:3-10; Petition, 6, 31; Reply, 5, 12

Why IBM is Wrong: General-Purpose Register

and machine states. It is noted that while an emulation environment preferably results in the setting of hardware or condition code elements in the target architecture, the present invention also contemplates the situation in which condition codes are generated and stored in locations other than condition code registers in the target machine. 60

'289 Patent, 1:57-62; Reply 10

Why IBM is Wrong: General-Purpose Register

code. To illustrate, refer to the tables presented in paragraphs [0025] and [0031] in applicants' specification. The target machine condition code, as a result of the sequence of instructions, will be held in register rC. An intermediate condition code value for that target machine condition code is determined (e.g. by performing the first four instructions in these examples) and placed in rC. That intermediate condition code value is a provisional value for the target machine condition code – that is, it is not the target machine condition code, *per se*, but rather a provisional value for the target machine condition code because it is subject to change based on the distinguishing information (the value stored in register rX in these examples). Based on performing the final instruction, in these examples, the target machine condition code is determined and stored in rC, which may or may not be a modified version of the intermediate condition code value that was determined and stored in rC prior to performance of the final instruction.

EX1004, 83-84; Petition 1, 14-15

Why IBM is Wrong: General-Purpose Register

LzLabs argues that “[t]he disclosed technique of the ’289 patent results in generating and storing an encoding of the condition code setting in register *rC*, ***a general-purpose register on the target machine.***” Reply, 4. LzLabs is mistaken. LzLabs’ made-up attorney argument is not supported by the evidence. Tellingly, the Reply does not point to the Specification to support LzLabs’ contention that *rC* is a “general-purpose register.” There is no such disclosure. Instead, the Specification discloses that an instruction executed in the target machine ***sets one or more target CPU flag bits*** and the result is placed in register *rC*. Accordingly, register *rC* holds the result of one or more target CPU flag bits. EX1001, 7:51-55

Sur-reply, 8-9

Why IBM is Wrong: General-Purpose Register

tions. First, an instruction (step 100 in FIG. 2) in the target machine's architecture is executed which mimics the instruction present in the source instruction stream in a manner which sets one or more target CPU flag bits and which places a result in a storage location (first location, such as rC above) accessible to the emulator. *Next, an instruction (step 105) is* 55

'289 Patent, 7:51-56

9. The computer program product of claim 8, wherein said first set of one or more possible outcomes comprises one case result and said second set of one or more possible outcomes comprises at least two other case results, and wherein said generating comprises:

mimicking an instruction present in a source instruction stream in a manner which sets one or more target central processing unit (CPU) flag bits and which stores a first result in a first storage location;

'289 Patent, Claim 9; Reply, 12, 19

rC Holds the Emulated Condition Code Value, Not “The Result of One or More Target CPU Flag Bits”

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

Target machine cc
in rC = emulated
IBM mainframe cc

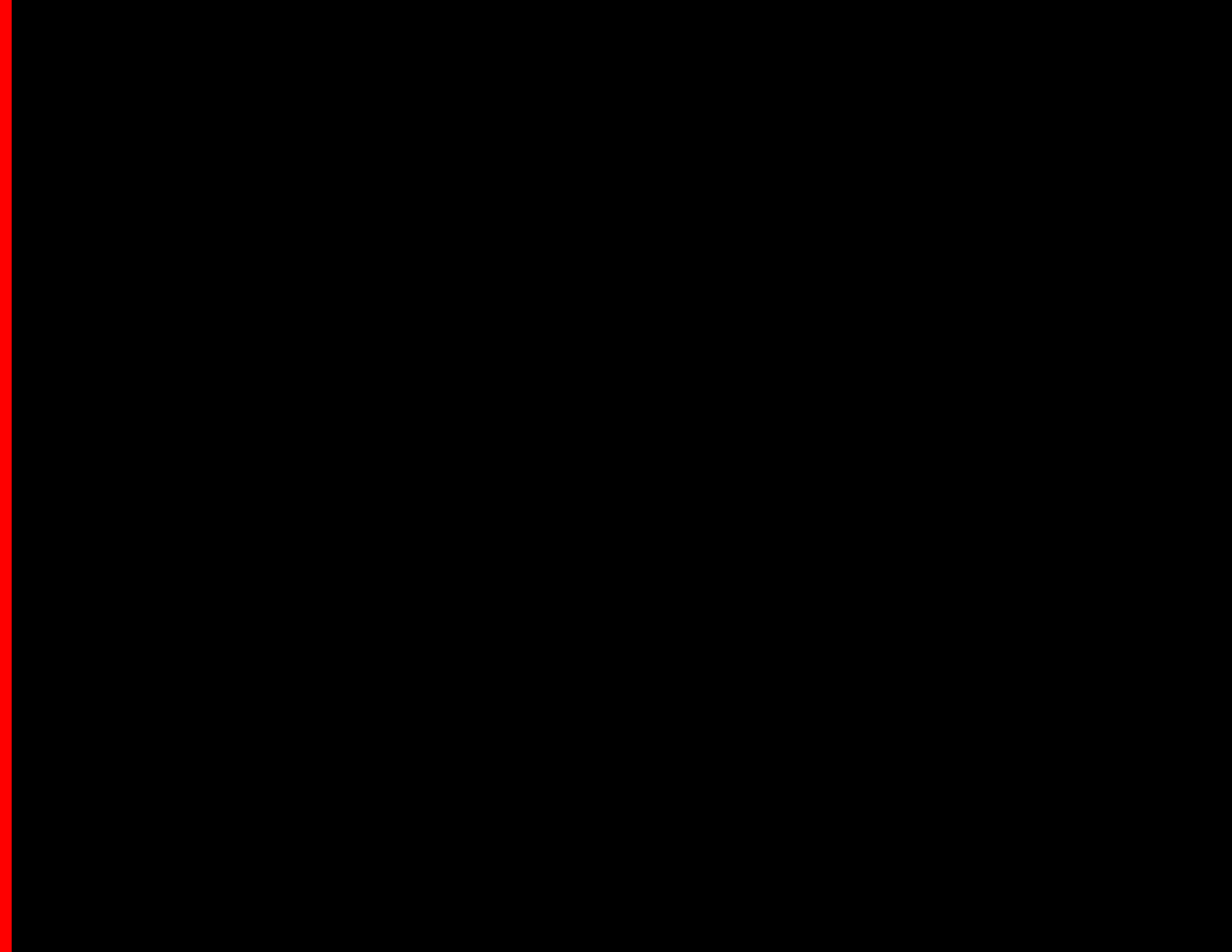
‘289 patent, 5:41-56; Petition, 5-7;
Reply 8-9, 12

IBM Fails to Offer a Construction of “Target Machine Condition Code” that Excludes Loderer

V. CLAIM CONSTRUCTION

In the Institution Decision, the Board determined that no claim terms require express construction. Institution Decision (Paper 7), 7. Solely for purposes of this IPR, IBM adopts the Board’s application of the plain and ordinary meaning of the claim terms. EX2001, ¶¶75-76.

POR, 26



The Challenged Claims Do Not Require “CPU flag bits”

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:

20

a non-transitory computer readable storage medium readable by a processor and storing instructions for execution by the processor for performing a method comprising:

obtaining by the target machine a computer instruction 25

from the source machine, said source 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 30

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 35

using 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 40

and subject to change based on the distinguishing information; and

determining, based on the intermediate condition code value and based on the distinguishing information, the target machine condition code, wherein 45

at least part of said distinguishing information is separate from the intermediate condition code value.

‘289 Patent, Claim 1;
Petition, 22-38; Reply 5-6

Distinguishing
information

Intermediate
condition code value

Target machine
condition code

The Challenged Claims Do Not Require “CPU flag bits”

Distinguishing
information in rX

Intermediate
cc value in rC

Target machine cc in
rC = emulated IBM
mainframe cc
generated by taking
difference of rX and
rC—[5] sub rC, rC, rX

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

COMPARE LOGICAL

Resulting Condition Code:

- 0 Operands equal
- 1 First operand low
- 2 First operand high
- 3 --

‘289 patent, 5:41-56;
EX1026, 78-79; Petition,
5-7; Reply 8-9, 12

The Challenged Claims Do Not Require “CPU flag bits”

Distinguishing
information in rX

30 After the execution of instruction [2] (set_bit_on_zero), the status of register rC is unchanged but the contents of rX are set equal to “1” if the two operands, rA and rB, are the same based on the contents of rC (limited to zero or not in this case). The execution of instruction [2] does not affect the contents of rC. Additionally, instruction [2] does not affect the CPU carry bit.

‘289 Patent, 6:28-32

Intermediate
cc value in rC

60 Instruction [4] (add_to_carry_immed) is then executed with the arguments shown (rC, rC, 0), with “0” being an immediate operand. With the operands shown, it carries out the operation: rC+“CPU carry bit”+0. While it also sets the CPU carry bit as well, this result is not required for subsequent processing. It is seen in Table II above that if rA>rB

Petition, 6; Reply, 4-5, 10

‘289 Patent, 6:59-64

“CPU flag bits” Are Not Required To Determine an Intermediate Condition Code Value

determining an intermediate condition code value, the intermediate condition code value being a provisional value for the target machine condition code 40 and subject to change based on the distinguishing information; and

‘289 Patent, 9:38-42
(excerpt of Claim 1)

IBM Relies on Extrinsic Evidence to Contend Condition Codes are “Special Bits”

As EX2004 discloses and Mr. Goodin explains, condition codes are “special bits set by ALU operations.” EX2004, 8; EX2001, ¶35. Exemplary architectures that implement condition codes include Intel’s 80x86 (e.g., IA32) architecture and Apple-IBM-Motorola’s PowerPC architecture. EX2001, ¶35.

POR, 6

35. As EX2004 discloses, condition codes are “special bits set by ALU operations.” EX2004, 8. Exemplary architectures that implement condition codes include Intel’s 80x86 (e.g., IA32) architecture and Apple-IBM-Motorola’s PowerPC architecture.

EX2001, ¶ 35

2004

Hennessy, John L., et al. *Computer Architecture: A Quantitative Approach*. India, Elsevier Science, 2006 (excerpt)

IBM Relies on Extrinsic Evidence to Contend Condition Codes are “Special Bits”

30. Condition codes are “special bits” that represent hardware state information used in computer architectures to hold information about the results of various hardware operations. EX2004, 8 (“[S]pecial bits set by ALU operations, possibly under program control.”); EX1005, 2 (“[A] special hardware flag, known as condition code or and usually comprising several bits,...”); EX2003, 76:22-23 (“[T]he condition code is typically implemented in special hardware.”). These

EX2001, ¶ 30

IBM Relies on Extrinsic Evidence to Contend Condition Codes are “Special Bits”

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. As a simple example, a special

‘289 Patent, 1:18-23; Petition, 3-4; Reply, 12, 23, 27)

Ex. 1001, 1:8–14. The ’289 patent explains that executing various operations may result in the generation of condition codes, which are bits of data indicating the outcome status of instruction execution. *Id.* at 1:18–23.

Institution Decision , 2

The POR fails to explain what makes bits of data “special” to qualify as a condition code or why the patent should be so limited

The '289 specification does not describe condition codes as “special bits,” and neither the POR nor PO’s expert explains what would make bits of data “special” to qualify as a condition code.⁷ Nor does the POR or PO’s expert explain why the bits stored in general purpose register rC of the disclosed embodiments are special enough to be condition codes, but the bit patterns BM stored in Loderer’s register are not.⁸

⁸ Remarkably, the POR and PO’s expert declaration omit any discussion of the preferred embodiments of the '289 patent or the disclosed examples of a “target machine condition code.”

Reply, 19-20

The Claimed Target Machine Condition Code is Not a Side-Effect

generating a sequence of target machine instructions which together operate to derive an encoding for a target machine condition code for the computer 30 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 35 using branch instructions, the directly calculating comprising:

‘289 patent, 9:28-37 (Claim 1 excerpt)

register. EX2004, 8. Condition registers are simple to implement, compared to condition codes. EX2004, 8. *See also*, EX1024, 4 (“Condition codes are difficult to implement primarily because they are an irregular structure. As opposed to registers, where the references of updates of objects are explicit, condition codes are updated as a side-effect.”) EX2001, ¶36.

POR, 6

IBM Relies on Extrinsic Evidence to Contend Condition Codes are “Special Bits”

Name	Examples	How condition is tested	Advantages	Disadvantages
Condition code (CC)	80x86, ARM, PowerPC, SPARC, SuperH	Tests special bits set by ALU operations, possibly under program control.	Sometimes condition is set for free.	CC is extra state. Condition codes constrain the ordering of instructions since they pass information from one instruction to a branch.
Condition register	Alpha, MIPS	Tests arbitrary register with the result of a comparison.	Simple.	Uses up a register.
Compare and branch	PA-RISC, VAX	Compare is part of the branch. Often compare is limited to subset.	One instruction rather than two for a branch.	May be too much work per instruction for pipelined execution.

EX2004, 8; POR, 10, 18, 29

Disclosed Embodiment of the Challenged Claims

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

'289 Patent, 5:41-56