EXHIBIT W

US005167023A

United States Patent [19]

de Nicolas et al.

[11] Patent Number: 5,167,023

[45] Date of Patent: Nov. 24, 1992

[54] TRANSLATING A DYNAMIC TRANSFER CONTROL INSTRUCTION ADDRESS IN A SIMULATED CPU PROCESSOR

[75] Inventors: Arturo M. de Nicolas; John C. O'Quin, III, both of Austin, Tex.

[73] Assignee: International Business Machines, Armonk, N.Y.

[21] Appl. No.: 625,780

[22] Filed: Dec. 7, 1990

Related U.S. Application Data

[63] Continuation of Ser. No. 151,137, Feb. 1, 1988, abandoned.

[51] Int. Cl.[5] G06F 9/30

[52] U.S. Cl. 395/375; 395/500; 364/DIG. 1; 364/232.3; 364/262.4; 364/247.6; 364/247; 364/247.7; 364/262.9

[58] Field of Search 395/500, 375, 775

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,347,565	8/1982	Kareda et al.	364/200
4,370,709	1/1983	Fosdick	364/200
4,587,612	5/1986	Fisk et al.	364/200
4,638,423	1/1987	Ballard	364/200
4,700,291	10/1987	Saito	364/200
4,727,484	2/1988	Saito	364/200
4,841,476	6/1989	Mitchell et al.	364/900

FOREIGN PATENT DOCUMENTS

0217068 4/1987 European Pat. Off. .

OTHER PUBLICATIONS

K. J. McNeley et al., "Emulating a Complex Instruction Set Computer with a Reduced Instruction Set Computer", IEEE Micro, Feb., 1987, vol. 7, No. 1, pp. 60–71.

Graham, C., "Amiga's Trump Card: IBM PC Emulation", AmigaWorld, vol. 1, No. 2, Nov./Dec., 1985, pp. 34–35.

"SoftPC", Insignia Solutions, Inc., ISI Soft PC Data Sheet Rev., 3.0, Jan. 87, 8 pages.

Warner, E., "Unix-Based Workstations to Run DOS", Info World, Jul. 6, 1987, p. 8.

May, C., "Mimic: A Fast System/370 Simulator", SIGPLAN, 1987, Proceedings of the ACM SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques, Jun. 87, pp. 1–13.

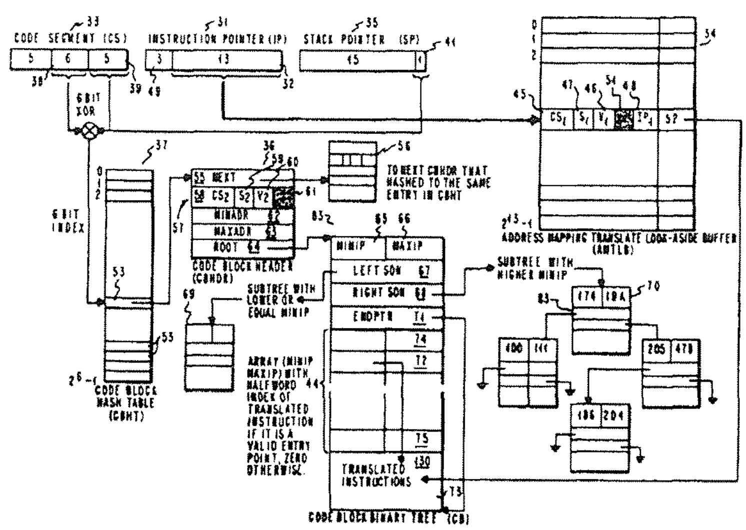
Primary Examiner—Kevin A. Kriess

Attorney, Agent, or Firm—Wayne P. Bailey; Marilyn D. Smith

[57] **ABSTRACT**

The system and method of this invention simulates the flow of control of an application program targeted for a specific instruction set of a specific processor by utilizing a simulator running on a second processing system having a second processor with a different instruction set. The simulator reduces the number of translated instructions needed to simulate the flow of control of the first processor instructions when translating the address of the next executable instruction resulting from a dynamic transfer of control, i.e., resulting from a return instruction. The simulator compares the address that is loaded at run time by the return instruction with the return address previously executed by that instruction. If the last return address matches, the location of the return is the same. If the last return does not match, a translate look-aside buffer is used to determine the address. If the translate look-aside buffer does not find the address, then a binary tree look up mechanism is used to determine the address of the next instruction after a return. The performance of the simulator is enhanced by utilizing the easiest approaches first in the chance that a translated instruction will result most efficiently.

7 Claims, 11 Drawing Sheets



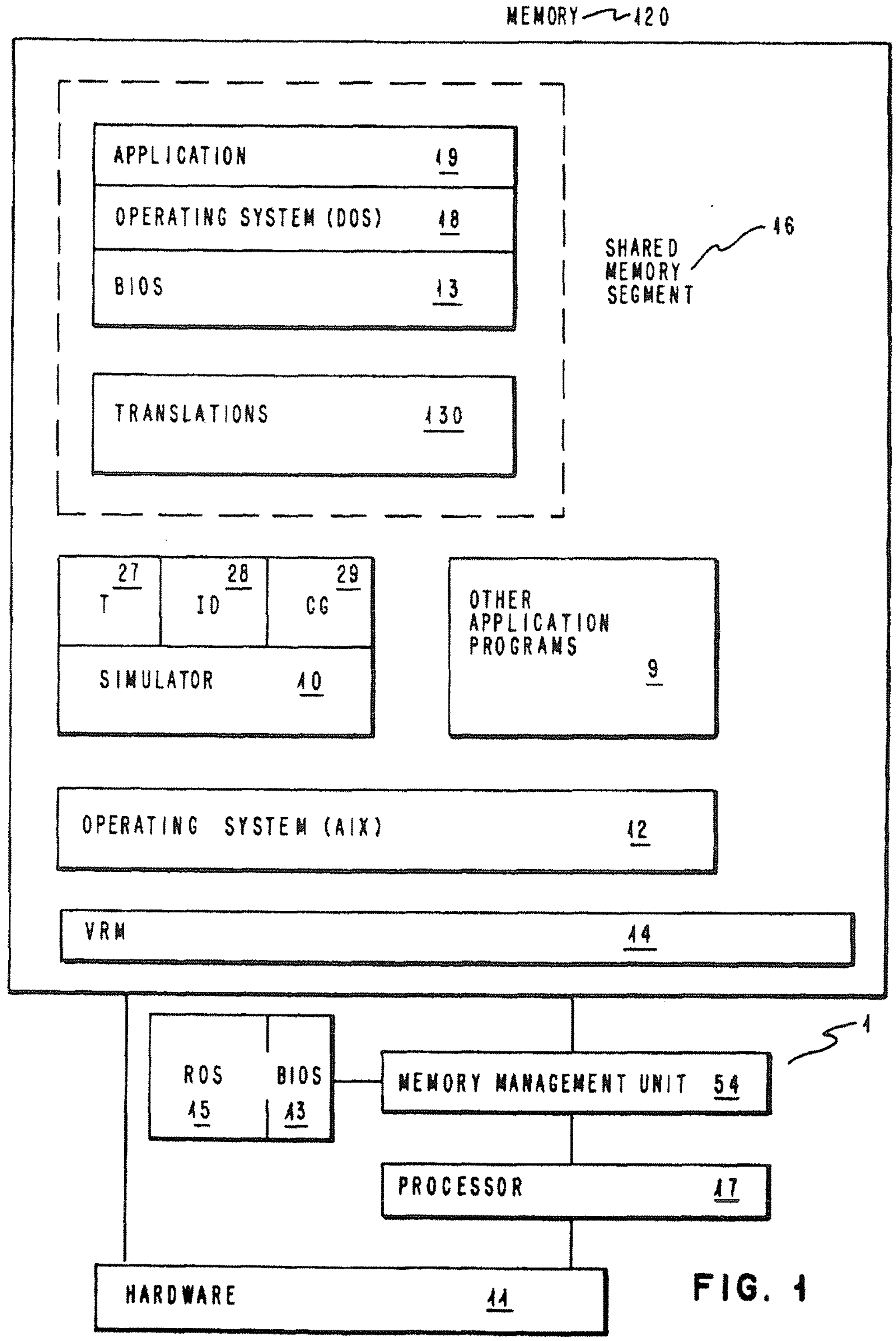

MEMORY 120
APPLICATION 19
OPERATING SYSTEM (DOS) 18
BIOS 13
SHARED MEMORY SEGMENT 16
TRANSLATIONS 130
T 27
ID 28
CG 29
SIMULATOR 10
OTHER APPLICATION PROGRAMS 9
OPERATING SYSTEM (AIX) 12
VRM 14
ROS 15
BIOS 13
MEMORY MANAGEMENT UNIT 54
1
PROCESSOR 17
HARDWARE 11

FIG. 1

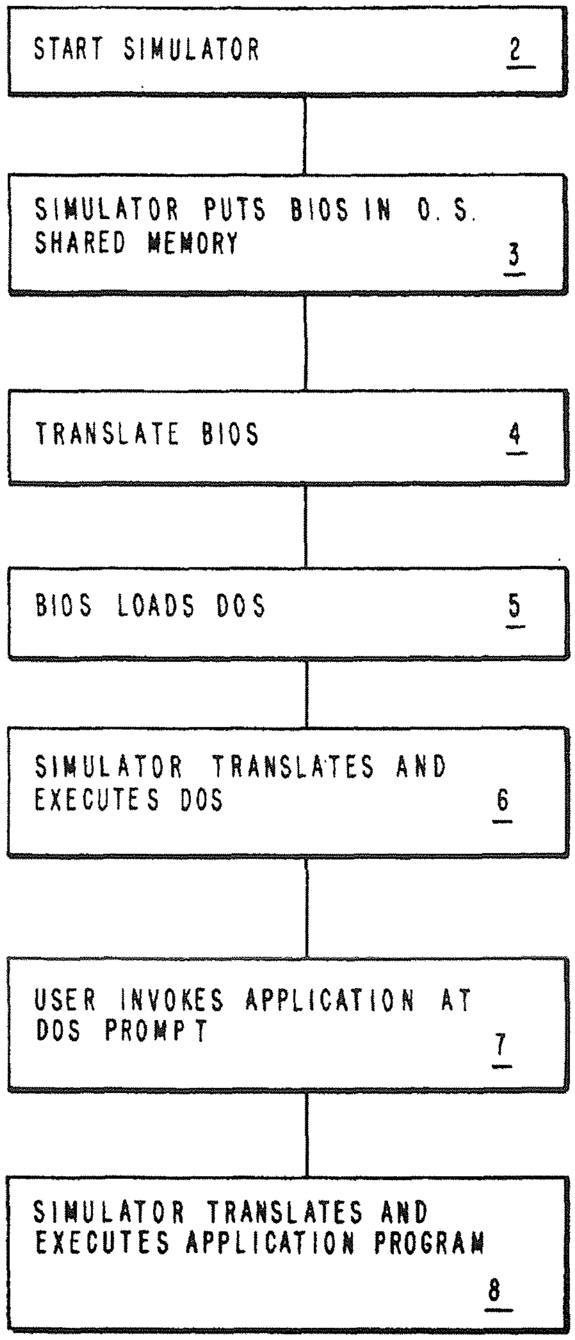
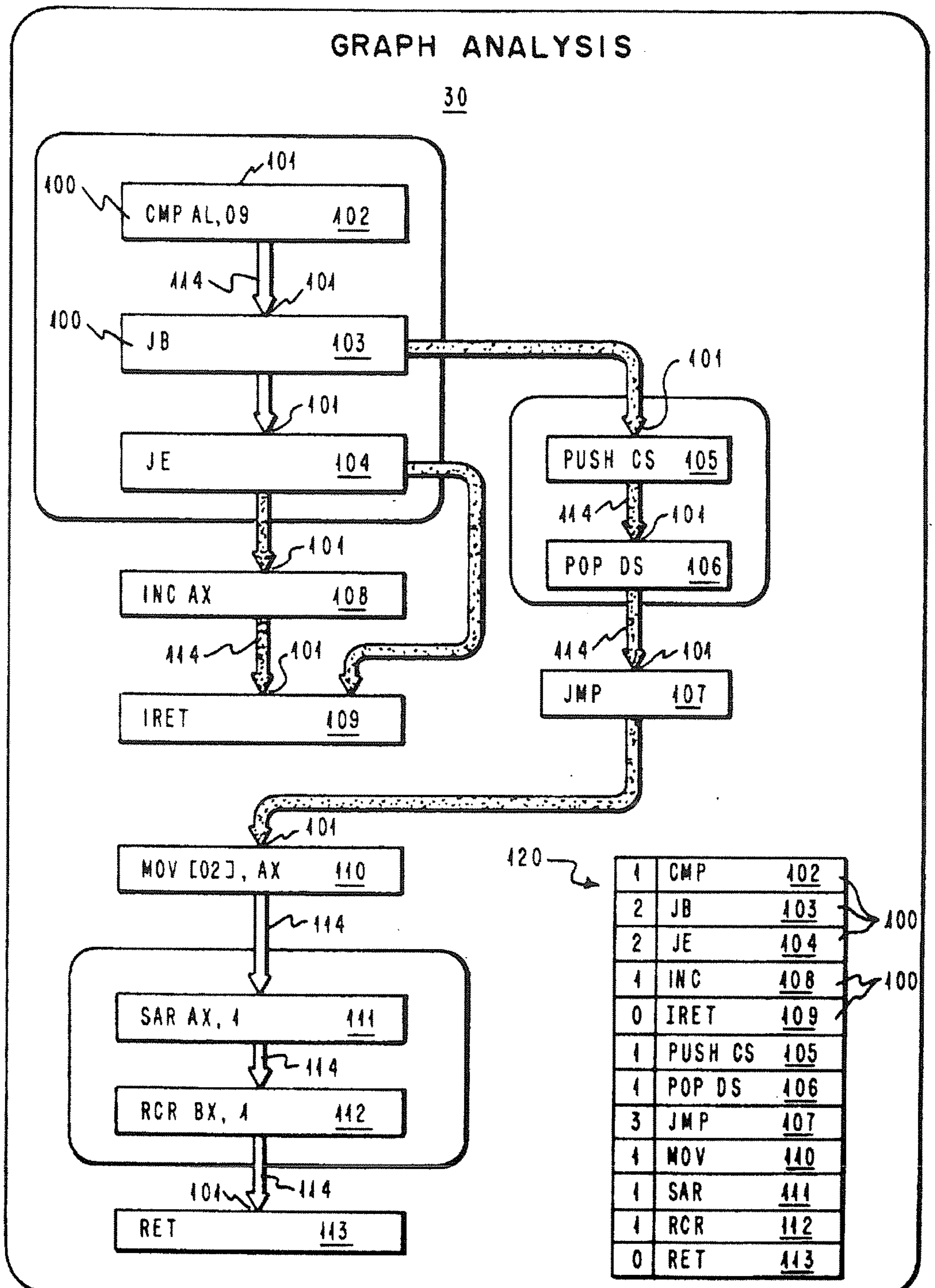

START SIMULATOR 2
SIMULATOR PUTS BIOS IN O.S. SHARED MEMORY 3
TRANSLATE BIOS 4
BIOS LOADS DOS 5
SIMULATOR TRANSLATES AND EXECUTES DOS 6
USER INVOKES APPLICATION AT DOS PROMPT 7
SIMULATOR TRANSLATES AND EXECUTES APPLICATION PROGRAM 8

FIG. 2

FIG. 3A


GRAPH ANALYSIS
30
100
101
CMP AL,09 102
114
101
100
JB 103
101
JE 104
101
PUSH CS 105
114
101
POP DS 106
101
INC AX 108
114
101
IRET 109
114
101
JMP 107
101
MOV [02], AX 110
114
SAR AX, 1 111
114
RCR BX, 1 112
101
114
RET 113
120
1 CMP 102
2 JB 103
2 JE 104
1 INC 108
0 IRET 109
1 PUSH CS 105
1 POP DS 106
3 JMP 107
1 MOV 110
1 SAR 111
1 RCR 112
0 RET 113
100
100

33 31

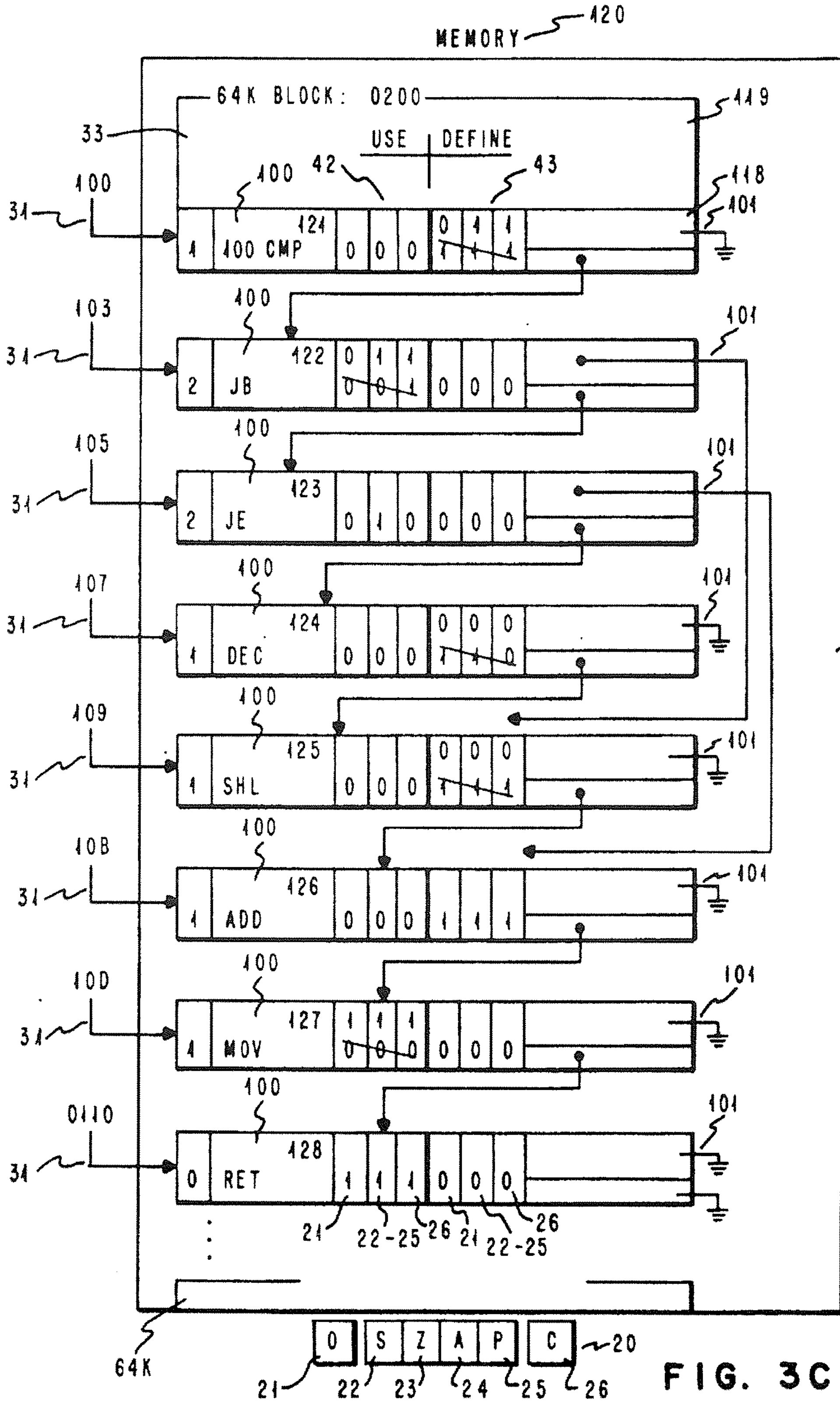
SP = EVEN

400

CS		IP			Ref.
0200	:	0100	CMP	AL, 9	421
0200	:	0103	JB	109	422
0200	:	0105	JE	10B	423
0200	:	0107	DEC	AX	424
0200	:	0109	SHL	BX, 1	425
0200	:	010B	ADD	BX, AX	426
0200	:	010D	MOV	[02], AX	427
0200	:	0110	RET		428

100

FIG. 3B



MEMORY
420
64K BLOCK: 0200
119
33
USE
DEFINE
400
42
43
118
101
31
100
1 100 CMP
121
103
2 JB
122
105
2 JE
123
107
1 DEC
124
109
1 SHL
125
10B
1 ADD
126
10D
4 MOV
127
0110
0 RET
128
50
21
22-25
26
64K
O
S
Z
A
P
C
20
22
23
24
25

FIG. 3C

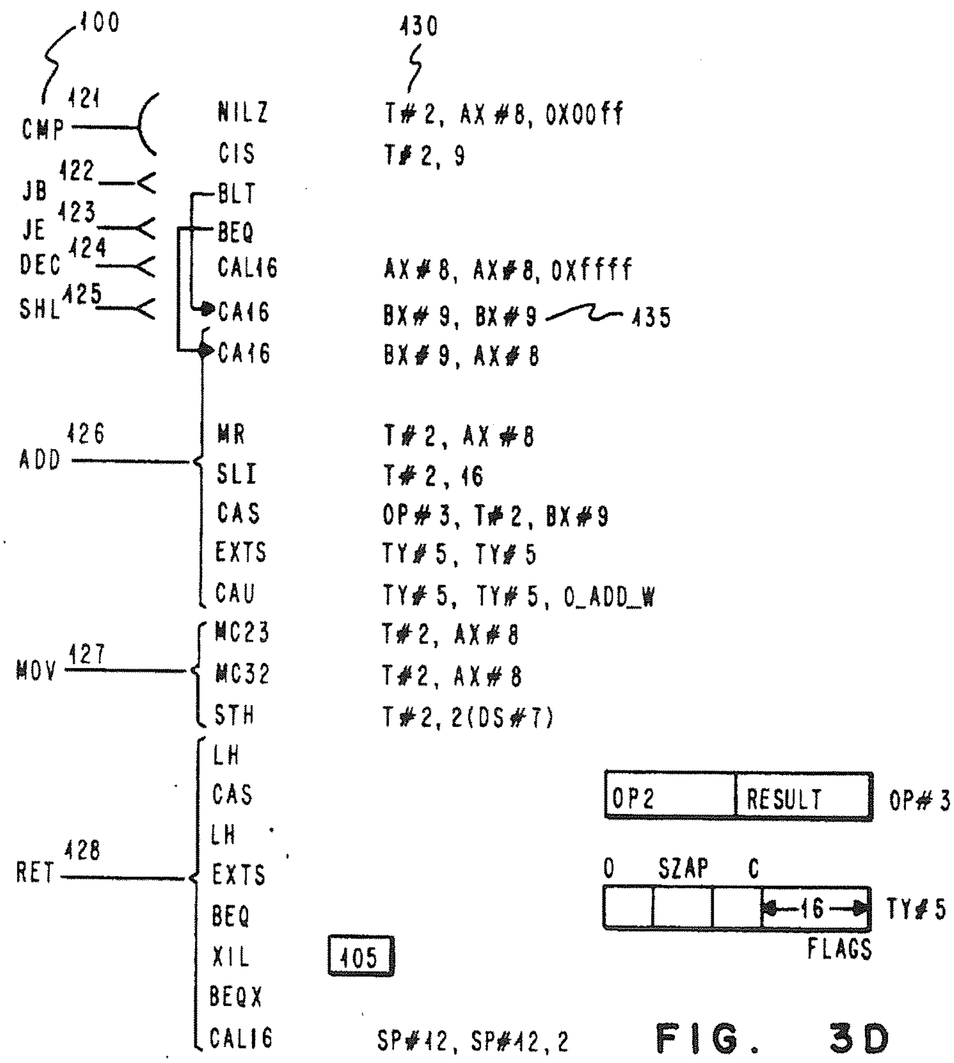


FIG. 3D

THE FOLLOWING INSTRUCTION SEQUENCE IS THEN USED ON EACH TRANSLATION OF A 80286 INSTRUCTION WHICH CAN MODIFY PC MEMORY:

1 CYCLE	NIUO R1, R2, 0x80FF	R2 = PC MEM ADDRESS IN SEGMENT 9 OR F	
		R1 <- ADDRESS OF STATUS BYTE	
2 CYCLE	LC R1, 0(R1)	R1 <- STATUS VALUE	
1 CYCLE	EXTS R1, R1	CHECK IF STATUS IS ZERO OR NON-ZERO	
1 CYCLE	BNE SUBROUTINE	JMP TO SUBROUTINE IF STATUS IS NON-ZERO	

FIG. 10

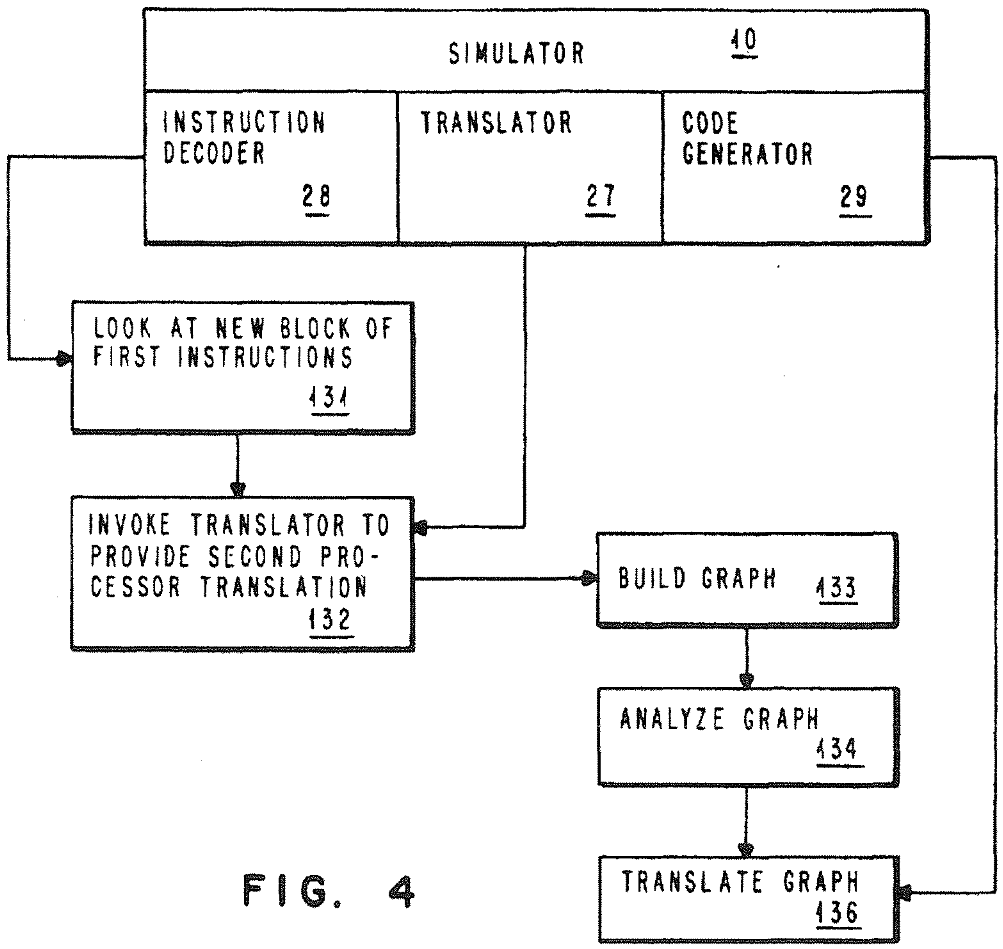


FIG. 4

```
XIL   t_1,t_0,PREVIOUS_IP# COMPARE BYTE REVERSED IP WITH PREVIOUS
BEQ   TARGET            # RELATIVE CONDITIONAL BRANCH IF THEY MATCH
```


FIG. 5

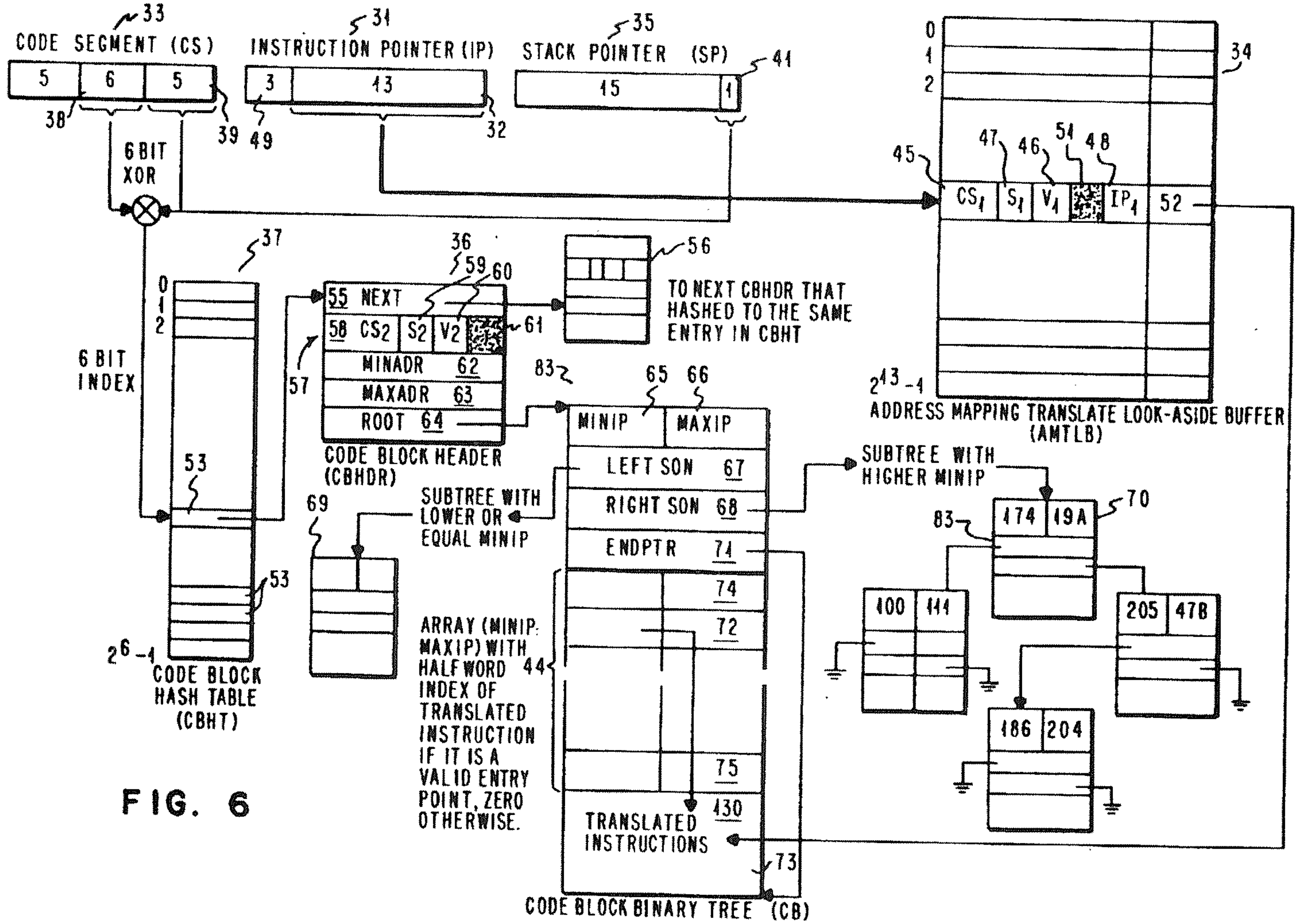

CODE SEGMENT (CS)
INSTRUCTION POINTER (IP)
STACK POINTER (SP)
6 BIT XOR
6 BIT INDEX
CODE BLOCK HASH TABLE (CBHT)
NEXT
MINADR
MAXADR
ROOT
CODE BLOCK HEADER (CBHDR)
TO NEXT CBHDR THAT HASHED TO THE SAME ENTRY IN CBHT
SUBTREE WITH LOWER OR EQUAL MINIP
MINIP
MAXIP
LEFT SON
RIGHT SON
ENDPTR
ARRAY (MINIP: MAXIP) WITH HALF WORD INDEX OF TRANSLATED INSTRUCTION IF IT IS A VALID ENTRY POINT, ZERO OTHERWISE.
TRANSLATED INSTRUCTIONS
CODE BLOCK BINARY TREE (CB)
SUBTREE WITH HIGHER MINIP
ADDRESS MAPPING TRANSLATE LOOK-ASIDE BUFFER (AMTLB)

FIG. 6

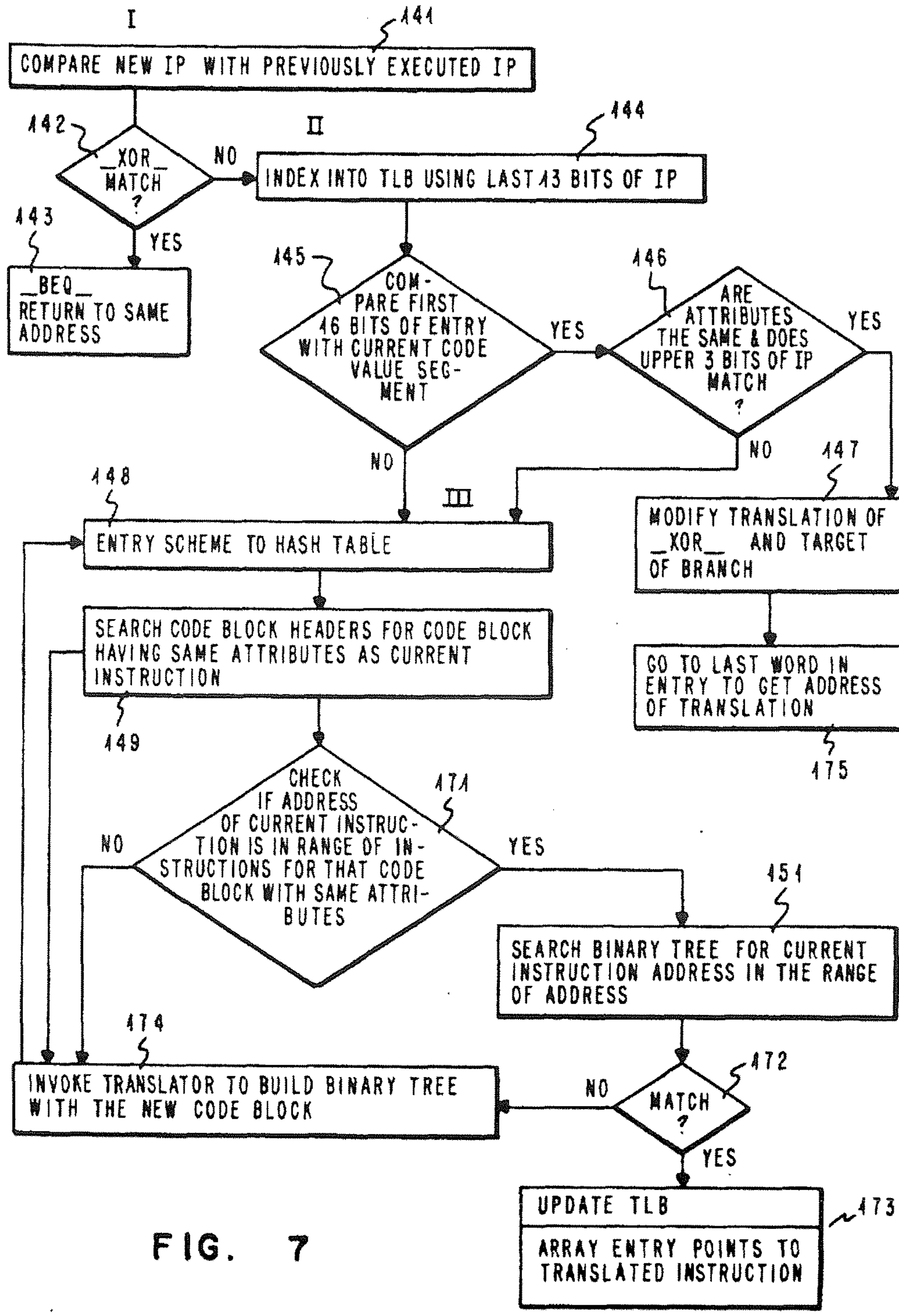

I
441
COMPARE NEW IP WITH PREVIOUSLY EXECUTED IP
442
XOR MATCH ?
NO
II
444
INDEX INTO TLB USING LAST 13 BITS OF IP
443
YES
BEQ RETURN TO SAME ADDRESS
445
COMPARE FIRST 16 BITS OF ENTRY WITH CURRENT CODE VALUE SEGMENT
YES
446
ARE ATTRIBUTES THE SAME & DOES UPPER 3 BITS OF IP MATCH ?
YES
NO
NO
447
448
III
ENTRY SCHEME TO HASH TABLE
MODIFY TRANSLATION OF _XOR_ AND TARGET OF BRANCH
SEARCH CODE BLOCK HEADERS FOR CODE BLOCK HAVING SAME ATTRIBUTES AS CURRENT INSTRUCTION
GO TO LAST WORD IN ENTRY TO GET ADDRESS OF TRANSLATION
449
175
CHECK IF ADDRESS OF CURRENT INSTRUCTION IS IN RANGE OF INSTRUCTIONS FOR THAT CODE BLOCK WITH SAME ATTRIBUTES
171
NO
YES
151
SEARCH BINARY TREE FOR CURRENT INSTRUCTION ADDRESS IN THE RANGE OF ADDRESS
174
INVOKE TRANSLATOR TO BUILD BINARY TREE WITH THE NEW CODE BLOCK
NO
MATCH ?
172
YES
UPDATE TLB
ARRAY ENTRY POINTS TO TRANSLATED INSTRUCTION
173

FIG. 7

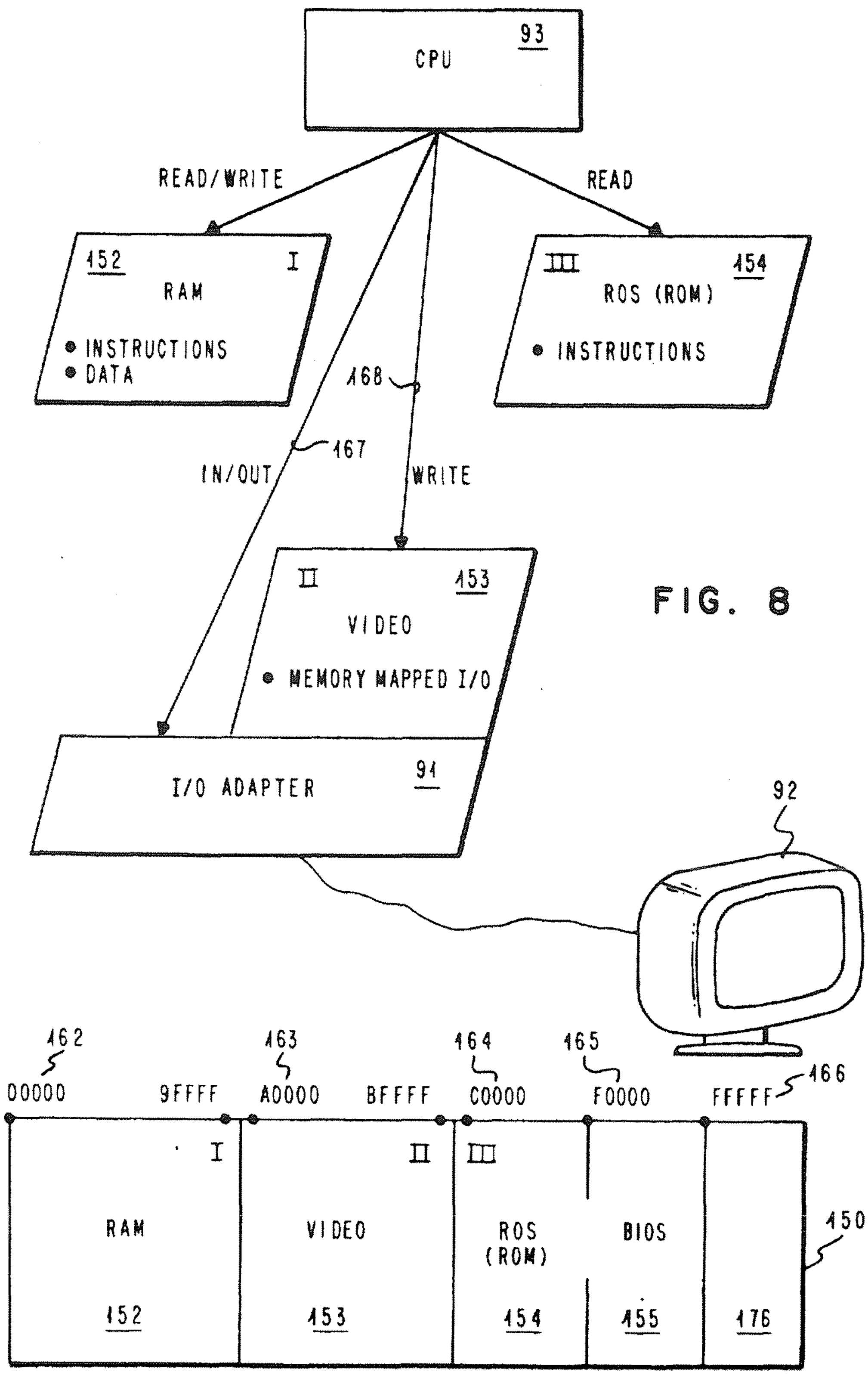

CPU
93
READ/WRITE
READ
152
I
RAM
• INSTRUCTIONS
• DATA
III
154
ROS (ROM)
• INSTRUCTIONS
168
167
IN/OUT
WRITE
II
153
VIDEO
• MEMORY MAPPED I/O
I/O ADAPTER
94
92
162
163
164
165
166
00000
9FFFF
A0000
BFFFF
C0000
F0000
FFFFF
I
II
III
RAM
152
VIDEO
153
ROS
(ROM)
154
BIOS
155
176
150

FIG. 8

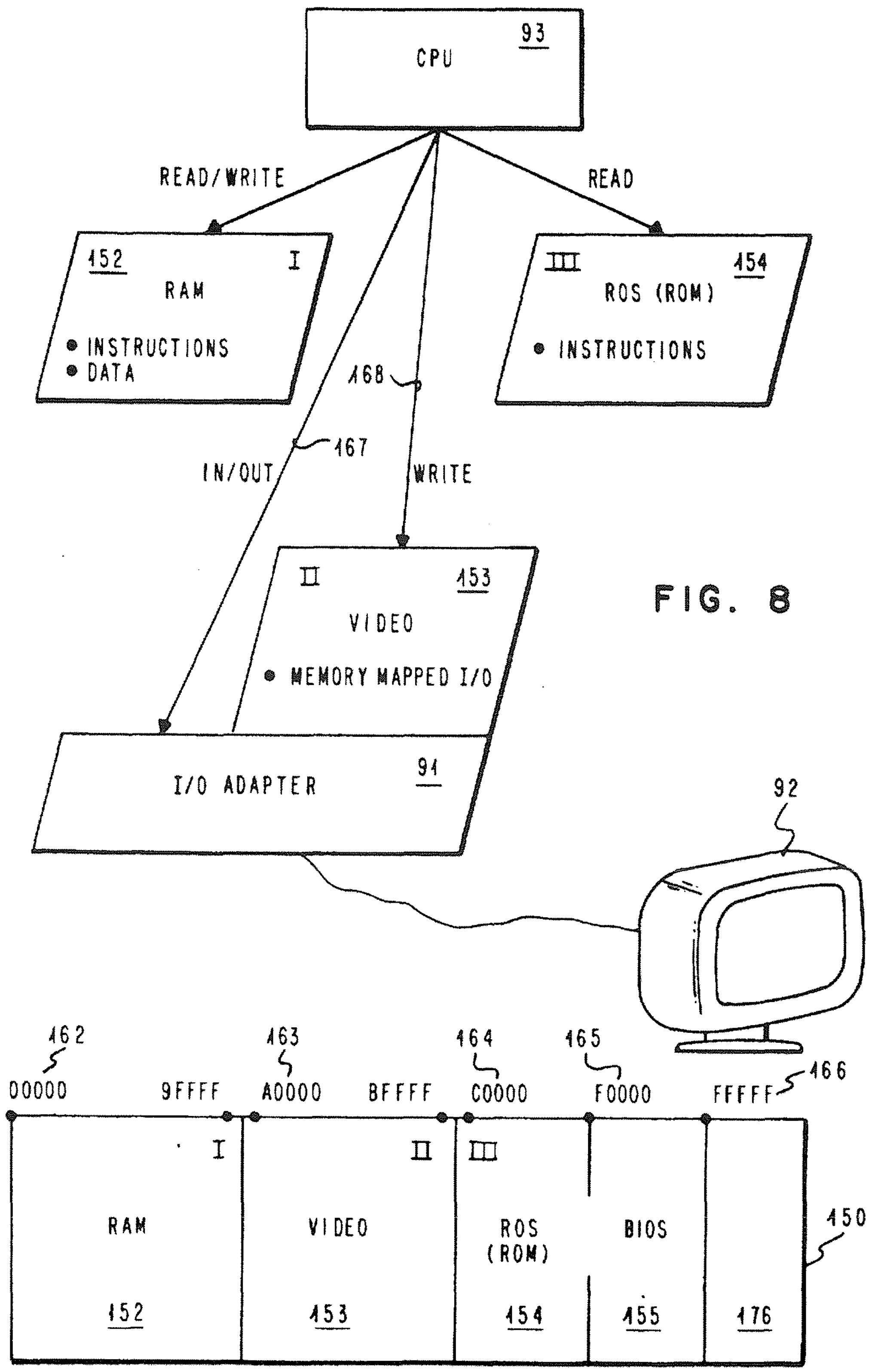

CPU
93
READ/WRITE
READ
152
I
RAM
• INSTRUCTIONS
• DATA
III
154
ROS (ROM)
• INSTRUCTIONS
168
167
IN/OUT
WRITE
II
153
VIDEO
• MEMORY MAPPED I/O
I/O ADAPTER
94
92
162
00000
9FFFF
163
A0000
BFFFF
164
C0000
165
F0000
FFFFF
166
I
RAM
152
II
VIDEO
153
III
ROS
(ROM)
154
BIOS
155
176
450

FIG. 8

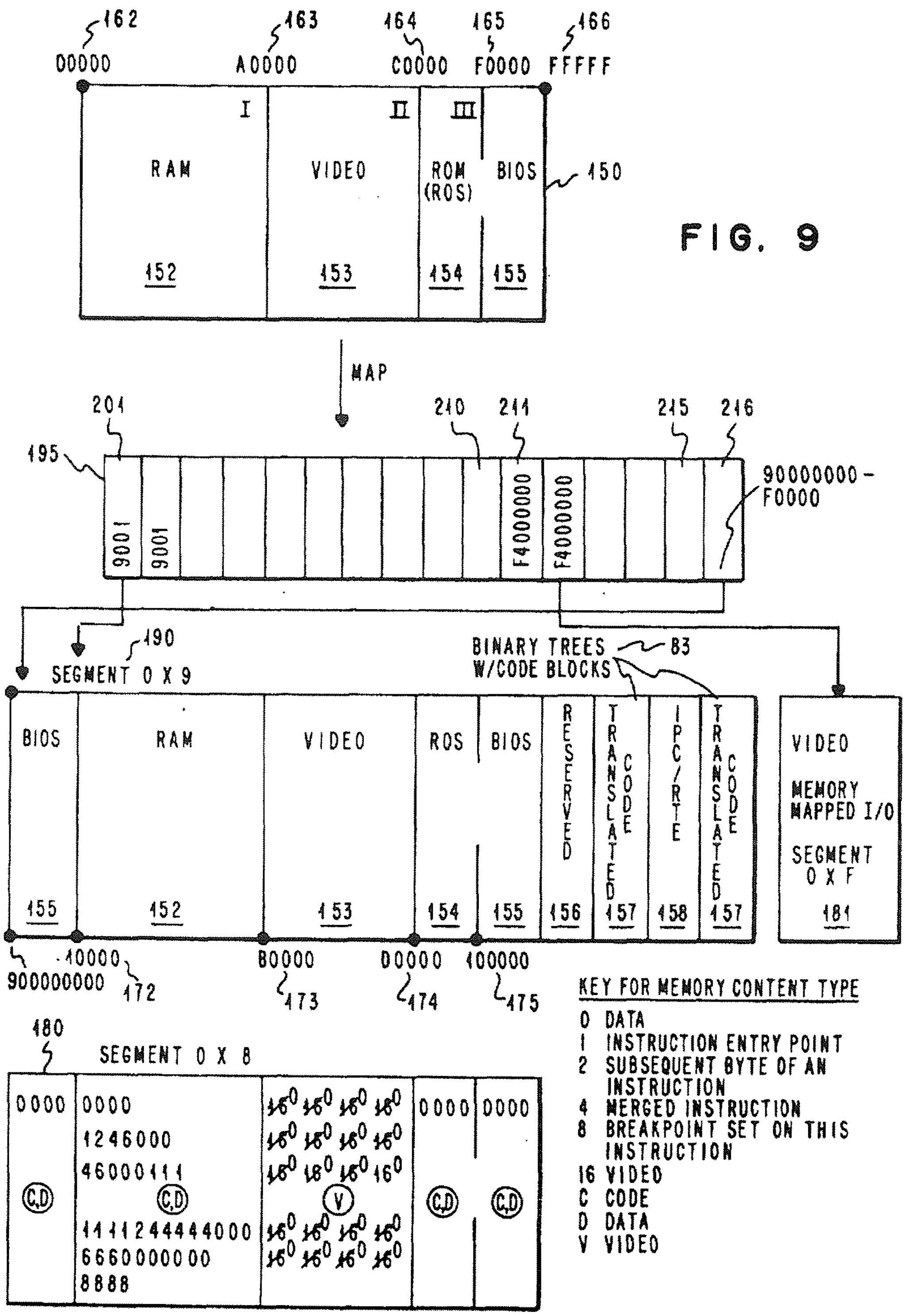


FIG. 9

KEY FOR MEMORY CONTENT TYPE

0 DATA
1 INSTRUCTION ENTRY POINT
2 SUBSEQUENT BYTE OF AN INSTRUCTION
4 MERGED INSTRUCTION
8 BREAKPOINT SET ON THIS INSTRUCTION
16 VIDEO
C CODE
D DATA
V VIDEO

TRANSLATING A DYNAMIC TRANSFER CONTROL INSTRUCTION ADDRESS IN A SIMULATED CPU PROCESSOR

This is a continuation of application Ser. No. 07/151,137, filed Feb. 1, 1988, now abandoned.

CROSS-REFERENCES TO RELATED APPLICATIONS

Ser. No. 820,451, filed Jan. 17, 1986 for a VIRTUAL TERMINAL SUBSYSTEM, currently co-pending, and assigned to the same assignee as the present invention.

Ser. No. 151,136, filed Feb. 1, 1988 for CONDITION CODE GRAPH ANALYSIS FOR SIMULATING A CPU PROCESSOR, now U.S. Pat. No. 4,951,195 and assigned to the same assignee as the present invention, which is hereby incorporated by reference.

Ser. No. 151,123, filed Feb. 1, 1988 for a SYSTEM AND METHOD FOR SIMULATING THE I/O OF A PROCESSING SYSTEM, now abandoned, and assigned to the same assignee as the present invention, which is hereby incorporated by reference.

Ser. No. 151,135, filed Feb. 1, 1988 for a MEMORY MAPPING AND SPECIAL WRITE DETECTION IN A SYSTEM AND METHOD FOR SIMULATING A CPU PROCESSOR, now abandoned, and assigned to the same assignee as the present invention, which is hereby incorporated by reference.

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention relates to data processing systems running applications written for a specific first processor of a first processing system, and more particularly to a system and method of simulating the first processor for running the applications on a second processing system having a second dissimilar processor.

2. Description of the Related Art

Current advances in computer technology have lead to ever changing processors, otherwise referred to herein as the central processing unit (CPU), of the processing system. Examples of the evolution of various processors are Intel's 8088 processor used in the IBM PC, Intel's 80286 processor used in the IBM PC AT[1], Intel's 80386 processor used in the IBM Personal System/2[2] model 80, and the IBM Research/ OPD Microprocessor (ROMP) which utilizes a Reduced Instruction Set Computer (RISC) architecture in the IBM RT PC[3]. Other processors include Motorola's 68000, 68020 among others.

The hardware of various processing systems changes rapidly to take advantage of the increased processing power of emerging processors. A disadvantage of changing hardware is that the software written for previous processors typically can not be used on the later hardware technology. In some cases where an application can be used on a different processing system other than the one it was originally written for, the performance of the application is not as good on the different processing system as it would have been on the processing system for which the application was originally written. As a result, software applications which may have had a long development cycle may become obsolete quickly. The demise of the earlier written software is all the more tragic when the function of the application as originally written is still very much pertinent and in demand on the new hardware processing systems.

As a result, there is typically only a limited amount of "new" software available that is specifically written for the "new" hardware design when the new hardware is initially released into the market place. This is due in part by the long development cycle of creating software application programs, and the confidentiality of the new hardware design by the manufacturer prior to releasing the hardware into the market place. The software manufacturer has to know certain facts about the hardware of a processing system before a software application program can be written for the processing system.

Ideally, a manufacturer of processing systems would like to have a vast amount of software available to run on the processing system as soon as the new hardware for the processing system is announced into the market place. A customer would more likely invest in a new processing system if the customer knows that an abundant supply of software is already available for use.

There have been several approaches in tapping the vast amount of software that has previously been written for "older" hardware designs. A previous hardware approach for being able to run applications originally written for another processor is to build the new processing system with a coprocessor. In this way, the processing system can run applications for both types of processors, the new processor and the old processor.

For example, the IBM RT PC contained an IBM PC AT coprocessor in order to use applications that were originally written for the IBM PC AT. However, since the coprocessor was supported at a low level in the operating system, the coprocessor could not take full advantage of the functions provided by the AIX[4] operating system. One of the functions provided by the AIX operating system is multi-tasking as described in co-pending application Ser. No. 820,451, filed Jan. 17, 1986 for a VIRTUAL TERMINAL SUBSYSTEM and assigned to the same assignee as the present invention, which is herein incorporated by reference.

The coprocessor, however, limits the user to one session at a time, since the coprocessor included a hardware adapter for emulating the PC AT. In other words, once the coprocessor was started, no other instances of the coprocessor could be running.

The coprocessor is also limited to the speed of the processor of the first processing system and cannot take advantage of faster second processing systems as they evolve.

A second approach is to simulate the second processor through software A software simulator provides a mechanism to run previously written software for one processor on a new processing system having a different processor. A software approach to simulation allows taking advantage of faster second processing systems as they evolve. It also allows the use of multitasking capabilities of the operating system to provide multiple instances of the first processor.

Some software simulators on the market today include SoftPC, by Insignia Solutions, and the Amiga Transformer by Simile Research Inc. for Commodore's Amiga (based on Motorola's 68000). Information on this system was published in the article "Amiga's Trump Card: IBM PC Emulation", *AMIGA WORLD*, Vol. 1, No. 2, November/December 1985. Phoenix Technologies also provided a simulator to simulate the Intel processor for the Apollo machine which has a Motorola 68000 processor.

Any specific CPU processor has a specific instruction set. When a software application program is developed for a specific CPU processor, it is compiled into object code. The object code is targeted to run on any CPU that supports the specific instruction set. A simulator takes object code that was written to run on a specific instruction set, and converts it to run on a different processor which may have a similar or different instruction set. The more the two instruction sets of the two processors are different, the more difficult it is to simulate the other processor.

For example, the Intel 80286 processor has a very rich instruction set in that it provides a wide variety of instructions. Each instruction is tailored specifically for a particular type of situation. Additionally, each instruction may be able to do several operations. In contrast, the ROMP processor in the RT PC has a reduced instruction set (RISC) processor which provides fewer instructions and less function per instruction. As each instruction in the Intel 80286 may be able to do several tasks, more instructions would be required with the ROMP RISC processor to accomplish the same tasks.

However, the speed of a processor can be increased by simplifying the instruction set. Although more instructions are required, no additional time is consumed on complicated instructions while executing the more common and simpler tasks.

Previous methods of software simulators created a subroutine that would simulate the effect of an instruction. Every time the machine being simulated needed to run that instruction, the subroutine would be called in order to decode and execute that instruction. The problem with this approach is that the overhead of decoding the instruction occurs every time the subroutine is called and executed. Consequently, the speed of the simulated processor is affected.

Instead of calling a subroutine each time an instruction needed to be executed, another software simulation approach compiled a shorter sequence of host machine instructions to simulate an instruction. As a result, the overhead of decoding and translating the instruction occurs only once, during the first time the instruction is encountered. This translation is then saved. From then on, every time that instruction is simulated, the translation is executed. This is often referred to as a second generation simulator. A first generation simulator will take an instruction one at a time and decode it in real time and execute it. The decoding is done for each instruction as each instruction is needed. A second generation simulator will go through the instructions one at a time, translate the instructions, and then reuse that translation instead of going back and translating again.

A previous second generation simulator was the simulator that simulated the IBM ROMP CPU on the IBM System/370 called RSIM. This simulator reserves a fixed amount of storage for each instruction (16 bytes for every half word) called cells. IBM 370 instructions would then be generated for each one of these cells for each RT instruction. If the amount of code generated is less than what would fit in one cell, which is usually the case, then it branches to the next boundary of the next cell. If the amount of code generated to simulate the instruction can not fit into one cell, then a subroutine call is generated that branches to a run time environment set of routines which performs the emulation and returns back to the cell to complete execution Another simulator simulates the processor of the IBM System/370 on the IBM RT PC which is described in the following article: May, C. "Mimic: A Fast System/370 Simulator", presented Jun. 11, 1987 at the Association of Computing Machinery Symposium on Interpreters and Interpretive Techniques, and published in SIGPLAN, 1987 proceedings of ACM.

A first generation simulator runs 50 to 100 host machine instructions per simulated instruction. A second generation simulator runs an average of 10 host machine instructions per simulated instruction.

If a simulator takes either 50 or 10 instructions to simulate one instruction on the simulated machine, the second processor running the simulator must be either 50 or 10 times as fast respectively as the simulated machine to be comparable in performance. It is therefore desirable to further reduce the number of simulator instructions per each simulated or translated instruction than what has previously been accomplished in the art.

For example, if a simulator could be designed to use only 4 instructions per simulated instruction, and the simulator processor is more than 4 times faster than the simulated machine processor, the simulator will be faster than the original machine being simulated. A user would then observe increased performance by using the simulated machine to run an application program than by using the machine for which the application program was originally written.

Therefore, the overall problem to overcome in simulating another processor is to further reduce the number of simulator (host) instructions per simulated instruction in order to increase the processing speed of the simulator.

SUMMARY OF THE INVENTION

It is therefore an object of this invention to reduce the average host machine instructions per simulated machine instructions.

The simulator of this invention runs applications originally written for a different processor by means of software emulation. The software approach of simulation allows the flexibility of utilizing the functions of the operating system of the simulator machine. In the preferred embodiment of this invention, the simulator runs as an application on the AIX operating system of the RT PC. It therefore can take advantage of the multitasking, multi-user capabilities of the AIX operating system to allow multiple applications originally written for the PC AT to run concurrently without any change to the application itself.

The method of simulation of this invention provides faster processing capability of the simulated processor than the previous methods of processor simulation by reducing the number of host machine instructions per simulated machine instruction. This was achieved by identifying key processing areas that had used more instructions than desired, and then by creating new methods to achieve the processing tasks through fewer instructions.

To increase the CPU simulation, i.e., reduce the average number of host instructions per simulated instruction, several key processing areas were identified that were currently utilizing more instructions than desired.

First, the processing task of maintaining and keeping the correct values of the condition codes was identified as disclosed in Ser. No. 07/151,136, filed Feb. 1, 1988 for CONDITION CODE GRAPH ANALYSIS FOR SIMULATING A CPU PROCESSOR, currently co-pending, and assigned to the same assignee as the present invention, which is hereby incorporated by reference. The method of this invention utilizes a graph analysis technique previously applied to compiler technology, and applies it to CPU simulator technology to dynamically determine whether condition codes will be needed by any subsequent instruction. The simulator saves sufficient information to generate those condition codes that may be needed. Otherwise, the number of translated instructions required are reduced for that instruction, if it is determined from the graph analysis that the condition codes are not needed.

Another area that was addressed to reduce the average number of instructions generated involves translating the addresses of instructions. The addresses of sequential instructions propose no great problem in translating since the translated address will also be the next sequential address. The difficulty arises in those instructions that are not common when viewing a program statically, but occur quite frequently in a dynamic mix of instructions. A primary example is the return from a subroutine instruction.

A processing system spends a comparatively large amount of time executing a return from subroutine. One difference with this type of instruction is that the address of the next instruction to be executed cannot be determined statically by just looking at the program. The program has to be actually running in order to determine the address to which that particular instruction will return. Most programs would probably return to the same place they had returned the last time that instruction had executed, as frequently occurs in a programmed loop. It is quicker for the simulator to compare the address that is loaded at run time by the return instruction with the return address previously executed by that instruction. If the last return address matches, the location of the return is the same. Only if it fails, are the more elaborate schemes of a translate look-aside buffer, and then binary trees, used as the look up mechanisms to determine the address of the next instruction after a return.

The three tier approach in determining the address of the next instruction simplifies the determination by using a compare in the case where the currently executing instruction returns to the same address as the last time it executed, by using a translate look-aside buffer if it returns to an address that was returned to at least once before, if not immediately before, and by using binary trees as a last resort if the preceding two cases fail. The return address is used in a general way to refer to an address of any instruction which transfers control by computing the next target address from a value in a register or memory.

A third prime processing area that was identified for reducing the average number of host instructions per simulated instruction was the processing task of being able to detect what happens when an instruction stores to memory as disclosed in Ser. No. 07/151,135, filed Feb. 1, 1988 for a MEMORY MAPPING AND SPECIAL WRITE DETECTION IN A SYSTEM AND METHOD FOR SIMULATING A CPU PROCESSOR, currently co-pending, and assigned to the same assignee as the present invention, which is hereby incorporated by reference. The simulator of this invention provides a method for checking any first processor instruction which updates memory, to determine if the instruction is modifying a subsequent instruction or performing a video buffer update. The method of this invention reduces the number of cycles, i.e. instructions, required to detect this modification.

BRIEF DESCRIPTION OF THE DRAWING

FIG. 1 is a block diagram showing the processing system environment of the preferred embodiment of this invention.

FIG. 2 is a flow chart showing the initial steps in starting, the simulator of this invention.

FIG. 3A shows a graph analysis of a sample flow of control of first processor instructions that are to be translated by the simulator of this invention.

FIG. 3B shows a second example of a flow of control of first processor instructions to be translated.

FIG. 3C illustrates a graph analysis of the condition codes in the flow of control of the first processor instructions shown in FIG. 3B.

FIG. 3D illustrates the flow of control of second processor instructions translated from the graph analysis of FIG. 3C.

FIG. 4 is a flow chart of the translation.

FIG. 5 is the program code used in the first method of the three tier approach for determining the translated instruction address of the next executable instruction.

FIG. 6 shows the data structures of the second and third approaches for determining the translated instruction address of the next executable instruction by mapping an instruction of one instruction set to a corresponding translation address of a simulator having a different instruction set.

FIG. 7 is a flow chart of the three tier approach for determining the translated instruction address of the next executable instruction.

FIG. 8 is a block diagram showing the types and contents of the memory of a processing system.

FIG. 9 illustrates the mapping of the memory of the first processing system into the memory of a second processing system, and a status table for indicating the type of contents of a store to memory.

FIG. 10 is the program code used to find the corresponding byte in the status table of a memory location in either shared memory or on an adapter.

DESCRIPTION OF THE PREFERRED EMBODIMENT

The preferred embodiment of the system and method of this invention simulates a processing system such as the IBM PC AT which utilizes an Intel 80286 processor having a complex instruction set, on a processing system 1 as shown in FIG. 1 such as the IBM RT PC which utilizes the ROMP processor having a reduced instruction set computer (RISC) technology. Although the RISC processor has less function per instruction, it can process instructions faster. The architectures between an Intel 80286 based machine and a RISC based machine are quite different from each other. The greater the difference there is between the architectures of two processing systems, the more difficult it is to simulate the one processor on the other processing system.

For more information on the RT PC processing system, the IBM PC AT processing system, and Intel's 80286 processor, the following references are suggested, and are hereby incorporated by reference. Bach,

M. J. *The Design of the UNIX Operating System*, Prentice Hall, 1986. Lang, T. G. and Mothersole, T. L., *Design of the RT PC VRM Nucleus*, Sep. 1, 1986. *AIX Operating System Commands Reference*, Version 2.1, IBM Corporation, SC23-0790. *AIX Operating System Managing the AIX Operating System*, Version 2.1, IBM Corporation, SC23-0793. *AIX Operating System Programming Tools and Interfaces*, Version 2.1, IBM Corporation, SC23-0789. *AIX Operating System Technical Reference*, Version 2.1, Volumes 1 and 2, IBM Corporation, SC23-0808 and SC23-0809. *IBM RT Personal Computer Technology*, IBM Corporation, SA23-1057, 1986. *Virtual Resource Manager Technical Reference*, Version 2.1, Volumes 1 and 2, IBM Corporation, SC23-0816 and SC23-0817, *iAPX 286 Programmer's Reference Manual Including the iAPX* 286 *Numeric Supplement*, Intel, 210498-003,1985, and *IBM PC AT Technical Reference Manual*, IBM Corporation, March 1984.

As shown in FIG. **1**, the simulator **10** runs as an application program on the operating system **12** of the processing system **1**. Referring to FIG. **2** in addition to FIG. **1**, as the simulator **10** is started, step **2**, the simulator **10** copies BIOS **13** containing 80286 instructions from read only storage (ROS) **15**, otherwise referred to as read only memory (ROM), into the operating system's **12** shared memory segment **16**, step **3**. The simulator **10** translates the BIOS **13**, step **4**, which loads the operating system (DOS) **18** for which the application **19** was originally written, step **5**. The simulator **10** then translates and executes that operating system **18**, step **6**. A user invokes the application **19** at the operating system prompt, step **7**, and the simulator **10** translates and executes the application program **19**, step **8**.

To increase the performance of the CPU simulation, i.e., reduce the average number of host instructions per simulated instruction, several key processing areas were identified that were currently utilizing more instructions than desired.

I. Condition Code Graph Analysis

First, the processing task of maintaining and keeping the correct values of the condition codes was identified.

Many processor instructions affect the flags register **20** (FIG. 3C) by updating the condition codes **21-26** in it to reflect the result of an operation. There are six different condition codes, the overflow flag **21**, the sign flag **22**, the zero flag **23**, arithmetic flag **24** also known as the half carry, the parity flag **25**, and the carry flag **26**. These condition codes **21-26** indicate common conditions such as whether the result was zero, whether it was negative, whether a carry out of a register occurred, or whether an overflow condition resulted. It also includes conditions that indicate the parity (even or odd) of the lower byte of the result and the carry out of the lower 4 bits of the operation (half carry).

To simulate a first processor instruction set by keeping an up to date version of the first processor flag register **20** would require additional cycles for every instruction that affects the register. This is especially true if the architecture of the first processor defines several different combinations of condition code updates. For example, the condition codes may be always set or cleared, computed, left unchanged, or left undefined.

A previous simulator RSIM, the RT PC processor simulator on IBM S/370, used a scheme to reduce the overhead of keeping track of the condition codes. Registers were reserved for this purpose. The reserved register contained the **32** bit values involved in an operation, and the type of operation performed. The idea was to postpone the work of determining the value of the condition code until an instruction that actually requires it is simulated. Nevertheless, there is still the overhead of saving away these values and types. This overhead is unnecessary if all possible subsequent paths contain an instruction which modifies the same condition codes without any intervening instruction needing them.

In order to determine which modifications of the flag register **20** actually were used by subsequent instructions, the simulator **10** of this invention relies on information provided by a graph analysis **30**, FIG. **3A**, or **50**, FIG. **3C** of a block of first processor instructions **100**. These techniques were previously applied in optimizing high level language compilers. However, it is believed that this is the first time that this technique has been applied to the problem of processor simulation.

When the simulator **10** reaches a new block of first processor instructions **100**, step **131**, FIG. **4**, the simulator **10** invokes the translator **27** to provide a second processor translation, step **132**. The translation occurs in three phases.

First, a graph **50**, FIG. 3C, is built, step **133**, FIG. **4**, which represents the structure of the block of first processor instructions **100**. Each node **101** in the graph corresponds to one instruction **100**. A first processor instruction decoder **28** fills in each node **101** with information about the instruction **100** including which registers and condition codes **21-26** the instruction **100** needs, block **42**, in order to execute, and which condition codes **21-26** the instruction **100** sets, block **43**, as a result of executing. It will be noted that the use register **42** and the set register **43** have a separate bit for the overflow flag **21** and the carry flag **26**, and groups the remaining condition codes **22-25** together in one bit. Therefore if any of these condition codes **22-25** are used or set by an instruction **100**, the middle bit of registers **42, 43** will so indicate.

Second, step **134**, FIG. **4**, the graph **30** is analyzed to determine where interrupts must be polled, how to order the translations so as to minimize branching, and which condition codes defined by an instruction are actually used.

Third, step **136**, FIG. **4**, the code generator **29** is invoked to translate the graph **30** into second processor instructions **130**, FIG. 3D.

At the time of code generation, step **136**, the information in the graph **50** will indicate if the condition codes **21-26** defined by an instruction are actually used. For example, in a majority of cases, the condition codes defined by a shift (SHL) instruction **125**, FIG. **3B**, **3C** are not actually used. The code generator **29** can use this knowledge and generate a single second processor instruction **135**, FIG. **3D** where four or five instructions may have been required to save the operands of the operation if the condition codes were needed. It is seen in the flow of control of the instructions **100** of FIG. **3B**, **3C** that the condition codes **21-26** for the ADD instruction **126** were needed in a subsequent instruction **128** in this example. Consequently, the translated instructions **130**, FIG. **3D**, resulted in six instructions instead of just one instruction had they not been needed. Nevertheless, should the graph analysis **50** indicate that the condition codes **21-26** are not needed, the extra translated instructions are not generated.

The simulator 10 translates first processor instructions 100 into second processor instructions 130, but does not do the translating one instruction at a time. The simulator 10 examines the first instruction that is about to be executed, and for which there is no translation, and continues examining subsequent instructions while building a graph 30 of those instructions as shown in FIG. 3A.

Each node 101 in the graph 30 corresponds to one first processor instruction 100. Each node 101 can have at most two descendants. In the case of sequential instructions 102, 105, 106, 108, 110, 111, 112, which transfer control only to the next sequential instruction in memory, the nodes 101 have only one descendant 103, 106, 107, 109, 111, 112, 113, respectively, as shown in FIG. 3A as a vertical line 114. There can be two descendants in the case of conditional branch instructions 103, 104, which test for a condition and branch to one instruction if the condition is true, and continues with the next instruction if the condition is false. It is possible for a node 101 to have no descendants, as is the case for the interrupt return instruction 109. The instruction 109 that has no descendants illustrates an instruction that transfers control dynamically. Also, a node 101 can have one descendant that is not a sequential instruction. The unconditional jump instruction 107 is an example of this.

As shown above, there are four types of instructions 100. These types are numbered in the code as follows. A node 101 is numbered "0" if the instruction at the node has no descendents, i.e., the return instruction 128 FIG. 3C, the interrupt return instruction 109, FIG. 3A, and the return instruction 113, FIG. 3A. A node 101 is numbered "1" if it is a sequential instruction and has one sequential descendent; e.g., compare instruction 121, decrement instruction 124, FIG. 3C, compare instruction 102, increment instruction 108, FIG. 3A. A node 101 is numbered "2" if the instruction has two descendents i.e., jump if below instruction 122, FIG. 3C, and 103, FIG. 3A. A node 101 is numbered "3" if the instruction is not a sequential instruction, but has only one descendent, i.e. jump instruction 107, FIG. 3A.

Having built a graph 50, FIG. 3C, that describes the block of first processor instructions 100, FIG. 3B, one node 101 per instruction 100, the simulator does an analysis of the graph 50. Each node 101 contains information on which condition codes 21-26 a first processor instruction 100 normally needs for execution, register 42, and which condition codes 21-26 are set, register 43, by that instruction after executing. In the case of the compare instruction 121, the compare instruction 121 does not need any condition codes 21-26 to execute, but it will set all of them, register 43. The jump if below (JB) instruction 122 needs the carry condition code 26 in order to execute because that is the condition being tested, register 42, but the JB instruction 122 does not set any condition codes after executing, register 43. The jump if equal (JE) 123 instruction needs the equal condition code bit in order to execute, but the JE instruction 123 does not set any condition codes after executing. The decrement instruction 124 sets the overflow flag 21 and the arithmetic flags 24, register 43, and leaves the carry condition code 26 unchanged, register 43. The graph analysis continues in this fashion for each of the rest of the instructions 100.

The nodes 101 are allocated in storage 120 sequentially as the application program 19 finds them. The search through the application program 19 originally written for the first processor is called a depth first search. The depth first search means that the instructions are searched sequentially until an end node (Type 0); e.g., the IRET instruction 109 FIG. 3A is reached. After a type zero node is reached, the search returns to the last instruction that had more than one descendent.

The order of instructions 100 FIG. 3A, stored in memory 120 would be compare 102, jump below 103, jump if equal 104, increment 108, and interrupt return 109. After the interrupt return 109, the search would go back to the jump if equal instruction 104 and examine the second descendent. The second descendent, interrupt return 109 is an instruction that is already stored in memory. Therefore, the search returns to the next previous node that had two descendents, the jump if below 103, and follows the second path. New code is then found in this path. The next instructions are stored in memory in order as shown in FIG. 3A.

Referring to FIG. 3B and 3C, for each of the nodes 101, the simulator keeps two fields in memory, one is the condition codes 21-26 that are needed by that instruction 100, register 42, the other is the condition codes 21-26 that are set by that instruction 100, register 43. At that point a propagation process is performed to optimize which condition codes must be set. This is accomplished by going from the last instruction allocated to the first instruction allocated. In this order, instructions are taken out of a circular queue. The register 42 of a node 101 is updated by ANDing the compliment of register 43 of that node with register 42 of all descendant nodes, and then ORing this result into the register 42 of a node 101 being updated. The updated register 42 reflects the descendants' needs for condition codes 21-26. If all descendants are marked done, then so is this node 101 being updated, and the node 101 is taken out of the queue. If the node is not done, the instruction is put at the end of the queue. The queue is gone through until the queue is empty indicating all of the nodes have been processed, or there are some remaining nodes in the queue that nothing has changed, i.e. a node has not been updated.

A second pass is performed through the graph. In this pass, the register 43 is updated by reducing the condition codes set to acknowledge those that are now shown as being needed by its descendants.

At the end of this analysis, each node 101 indicates which condition codes 21-26 must be set, register 43. The number of condition codes 21-26 could be less than the number initially set by that instruction. The set of condition codes will include only those condition codes that will be used by subsequent instructions. For example, the return instruction 128 indicates in register 42 that it will use all of the condition codes 21-26 as a precautionary measure since the subsequent instructions are not known until execution. The move instruction 127 typically does not use, register 42, the condition codes 21-26, but indicates a use, register 42, since the subsequent return instruction 128 needs them. The add instruction 126 does not use the condition codes 21-26, register 42, but sets all of them, register 43, which then can be used subsequently. The shift instruction 125 normally sets, register 43, all of the condition codes 21-26, but does not need to in this case since the subsequent add instruction 126 does not use them. It was already analyzed that the add instruction 126 would set the condition codes 21-26 for subsequent use. This analysis for updating the use register 42 and the set register 43 continues in reverse order through the list of instruc-

The simulator 10 translates first processor instructions 100 into second processor instructions 130, but does not do the translating one instruction at a time. The simulator 10 examines the first instruction that is about to be executed, and for which there is no translation, and continues examining subsequent instructions while building a graph 30 of those instructions as shown in FIG. 3A.

Each node 101 in the graph 30 corresponds to one first processor instruction 100. Each node 101 can have at most two descendants. In the case of sequential instructions 102, 105, 106, 108, 110, 111, 112, which transfer control only to the next sequential instruction in memory, the nodes 101 have only one descendant 103, 106, 107, 109, 111, 112, 113, respectively, as shown in FIG. 3A as a vertical line 114. There can be two descendants in the case of conditional branch instructions 103, 104, which test for a condition and branch to one instruction if the condition is true, and continues with the next instruction if the condition is false. It is possible for a node 101 to have no descendants, as is the case for the interrupt return instruction 109. The instruction 109 that has no descendants illustrates an instruction that transfers control dynamically. Also, a node 101 can have one descendant that is not a sequential instruction. The unconditional jump instruction 107 is an example of this.

As shown above, there are four types of instructions 100. These types are numbered in the code as follows. A node 101 is numbered "0" if the instruction at the node has no descendents, i.e., the return instruction 128 FIG. 3C, the interrupt return instruction 109, FIG. 3A, and the return instruction 113, FIG. 3A. A node 101 is numbered "1" if it is a sequential instruction and has one sequential descendent; e.g., compare instruction 121, decrement instruction 124, FIG. 3C, compare instruction 102, increment instruction 108, FIG. 3A. A node 101 is numbered "2" if the instruction has two descendents i.e., jump if below instruction 122, FIG. 3C, and 103, FIG. 3A. A node 101 is numbered "3" if the instruction is not a sequential instruction, but has only one descendent, i.e. jump instruction 107, FIG. 3A.

Having built a graph 50, FIG. 3C, that describes the block of first processor instructions 100, FIG. 3B, one node 101 per instruction 100, the simulator does an analysis of the graph 50. Each node 101 contains information on which condition codes 21-26 a first processor instruction 100 normally needs for execution, register 42, and which condition codes 21-26 are set, register 43, by that instruction after executing. In the case of the compare instruction 121, the compare instruction 121 does not need any condition codes 21-26 to execute, but it will set all of them, register 43. The jump if below (JB) instruction 122 needs the carry condition code 26 in order to execute because that is the condition being tested, register 42, but the JB instruction 122 does not set any condition codes after executing, register 43. The jump if equal (JE) 123 instruction needs the equal condition code bit in order to execute, but the JE instruction 123 does not set any condition codes after executing. The decrement instruction 124 sets the overflow flag 21 and the arithmetic flags 24, register 43, and leaves the carry condition code 26 unchanged, register 43. The graph analysis continues in this fashion for each of the rest of the instructions 100.

The nodes 101 are allocated in storage 120 sequentially as the application program 19 finds them. The search through the application program 19 originally written for the first processor is called a depth first search. The depth first search means that the instructions are searched sequentially until an end node (Type 0); e.g., the IRET instruction 109 FIG. 3A is reached. After a type zero node is reached, the search returns to the last instruction that had more than one descendent.

The order of instructions 100 FIG. 3A, stored in memory 120 would be compare 102, jump below 103, jump if equal 104, increment 108, and interrupt return 109. After the interrupt return 109, the search would go back to the jump if equal instruction 104 and examine the second descendent. The second descendent, interrupt return 109 is an instruction that is already stored in memory. Therefore, the search returns to the next previous node that had two descendents, the jump if below 103, and follows the second path. New code is then found in this path. The next instructions are stored in memory in order as shown in FIG. 3A.

Referring to FIG. 3B and 3C, for each of the nodes 101, the simulator keeps two fields in memory, one is the condition codes 21-26 that are needed by that instruction 100, register 42, the other is the condition codes 21-26 that are set by that instruction 100, register 43. At that point a propagation process is performed to optimize which condition codes must be set. This is accomplished by going from the last instruction allocated to the first instruction allocated. In this order, instructions are taken out of a circular queue. The register 42 of a node 101 is updated by ANDing the compliment of register 43 of that node with register 42 of all descendant nodes, and then ORing this result into the register 42 of a node 101 being updated. The updated register 42 reflects the descendants' needs for condition codes 21-26. If all descendants are marked done, then so is this node 101 being updated, and the node 101 is taken out of the queue. If the node is not done, the instruction is put at the end of the queue. The queue is gone through until the queue is empty indicating all of the nodes have been processed, or there are some remaining nodes in the queue that nothing has changed, i.e. a node has not been updated.

A second pass is performed through the graph. In this pass, the register 43 is updated by reducing the condition codes set to acknowledge those that are now shown as being needed by its descendants.

At the end of this analysis, each node 101 indicates which condition codes 21-26 must be set, register 43. The number of condition codes 21-26 could be less than the number initially set by that instruction. The set of condition codes will include only those condition codes that will be used by subsequent instructions. For example, the return instruction 128 indicates in register 42 that it will use all of the condition codes 21-26 as a precautionary measure since the subsequent instructions are not known until execution. The move instruction 127 typically does not use, register 42, the condition codes 21-26, but indicates a use, register 42, since the subsequent return instruction 128 needs them. The add instruction 126 does not use the condition codes 21-26, register 42, but sets all of them, register 43, which then can be used subsequently. The shift instruction 125 normally sets, register 43, all of the condition codes 21-26, but does not need to in this case since the subsequent add instruction 126 does not use them. It was already analyzed that the add instruction 126 would set the condition codes 21-26 for subsequent use. This analysis for updating the use register 42 and the set register 43 continues in reverse order through the list of instructions **100** previously stored in a specified order to perform the graph analysis.

As seen in FIG. 3D, the translation **130** required only one instruction each for the decrement instruction **124** and the shift instruction **125**, FIG. 3C. In contrast, the add instruction **126** that set the condition codes **21-26** for subsequent use took six instructions.

Likewise, the graph analysis **30** of FIG. 3A would also show that the compare instruction, which sets all of the condition codes, only needs two of the six condition codes in subsequent instructions, the zero bit used by the jump if equal (JE) instruction **104**, and the carry bit used by the jump if below (JB) instruction **103**. None of the other condition codes are needed. Furthermore, no descendent of the conditional branches shown in FIG. 3A needs the other condition codes. Therefore the simulator can translate the block of the three instructions **102**, **103**, **104** as a single unit without having to be concerned with setting the condition codes. Therefore the number of instructions that require the preservation of the corresponding condition codes are reduced.

By applying the previous compiler technique of graph analysis to a processor simulator, essential knowledge is gained about the use of condition codes by subsequent instructions. The code generator **29** which translates first processor instructions **100** into second processor instructions **130** uses the condition code information to produce in many cases a single second processor translated instruction. This reduces the unnecessary cycles if the flags register had been kept up to date after every instruction, and improves the performance of the simulator.

Besides utilizing the results from a graph analysis to obtain fewer translated instructions to simulate the flow of control, the results from a graph analysis can be used in the process of minimization of interrupt polling as disclosed in Ser. No. 07/151,123, filed Feb. 1, 1988 for A SYSTEM AND METHOD FOR SIMULATING THE I/O OF A PROCESSING SYSTEM, currently co-pending, and assigned to the same assignee as the present invention, which is hereby incorporated by reference.

II. Instruction Address Translation

The simulator in the preferred embodiment of this invention translates the instructions of Intel's iAPX 0286 processor (first processor) used in the IBM PC AT into simulator instructions of the ROMP processor (second processor) used in the IBM RT PC. These translations are saved for reuse when the application program originally written for the simulated first processor transfers control to that same address again. For instructions that do not transfer control, determining the next instruction pointer (IP) is accomplished by adding the length of the instruction to the instruction's instruction pointer (IP). A similar sequence of sequential simulator instructions are generated by the simulator. Because the instructions follow sequentially, locating the corresponding translations is not required.

Another class of instructions transfers control statically. That is, the new instruction pointer is computed by adding or subtracting a fixed displacement from the instruction's pointer. These are known as relative branches. The simulator generates second processor relative branches to the corresponding translation.

The instruction set contains three instructions that transfer control within a code segment to a new instruction pointer which cannot be determined statically since it is loaded from a register or memory at run time. The RETurn from subroutine instruction is one of these. The speed with which this instruction is simulated affects the overall performance of the simulator, especially if a processing system spends more time executing a RETurn instruction than any other instruction. The other two instructions are the JuMP indirect and the CALL indirect (register or memory). They will be treated the same as the RETurn instruction.

For example, a first processor addresses instructions by using two registers. The code segment register **33**, FIG. 3B describes the location of a 64K block **119** of memory **120**, FIG. 3C. The instruction pointer register **31**, FIG. 3B, is an offset into that code segment **33**. The instruction pointer **31** describes at which of the 64K bytes the instruction **100** is located. The first processor addresses instructions by indexing into a code segment **33** with the instruction pointer **31**, FIG. 3C.

The simulator of this invention uses the data structures shown in FIG. 6 to map first processor instruction addresses **100** which consist of a code segment **33** and an instruction pointer **31** to the address in memory **120** of the corresponding second processor where a sequence of second processor instructions **130** perform the same function.

In the method of this invention, a new instruction pointer **31**, and code segment **33**, whose values are determined at run time, are converted into the simulator machine (second processor) address of the corresponding translation for those instructions that transfer control dynamically, such as the RETurn from subroutine instruction, JuMP indirect, and the CALL indirect, or software interrupt instructions.

The simulator uses a three tier approach (FIG. 7) to simulate these three instructions that transfer control dynamically. The three tier approach is organized in succession. The fastest and most likely case is performed first. The second approach is used should the first approach fail. The third approach is the slowest approach which guarantees a successful conversion of the instruction pointer to an address of the simulator machine. FIG. 6 illustrates the second and third approaches.

Referring to FIGS. 5 and 7, the first operation performed on the new instruction pointer is to compare it with the value produced by the previous execution of that instruction, step **141**. If the values match, a simulator machine relative branch is performed to the corresponding address. An exclusive OR operation determines if the values match, step **142**, and a conditional branch transfers control if they do, step **143**. This allows for a fast computation of the look up address.

Referring to FIGS. 6 and 7, should the above approach fail, i.e, the instruction pointer **31** is different from the previous time that particular instruction had executed, a table look up is performed using the address mapping translate look-aside buffer **34** which is similar to a hardware translate look aside buffer. The conversion from a first processor instruction to a second processor instruction uses the method described below.

Second processor instructions **130**, when translated, assume certain values of first processor registers, called attributes. These attributes can be used by the translator to generate more efficient code in certain cases. For example, if the stack alignment is even, half word instructions can be used to transfer data to and from the stack. Otherwise, two separate byte instructions must be used. The value of the code segment **33** and the aligntions **100** previously stored in a specified order to perform the graph analysis.

As seen in FIG. 3D, the translation **130** required only one instruction each for the decrement instruction **124** and the shift instruction **125**, FIG. 3C. In contrast, the add instruction **126** that set the condition codes **21-26** for subsequent use took six instructions.

Likewise, the graph analysis **30** of FIG. 3A would also show that the compare instruction, which sets all of the condition codes, only needs two of the six condition codes in subsequent instructions, the zero bit used by the jump if equal (JE) instruction **104**, and the carry bit used by the jump if below (JB) instruction **103**. None of the other condition codes are needed. Furthermore, no descendent of the conditional branches shown in FIG. 3A needs the other condition codes. Therefore the simulator can translate the block of the three instructions **102**, **103**, **104** as a single unit without having to be concerned with setting the condition codes. Therefore the number of instructions that require the preservation of the corresponding condition codes are reduced.

By applying the previous compiler technique of graph analysis to a processor simulator, essential knowledge is gained about the use of condition codes by subsequent instructions. The code generator **29** which translates first processor instructions **100** into second processor instructions **130** uses the condition code information to produce in many cases a single second processor translated instruction. This reduces the unnecessary cycles if the flags register had been kept up to date after every instruction, and improves the performance of the simulator.

Besides utilizing the results from a graph analysis to obtain fewer translated instructions to simulate the flow of control, the results from a graph analysis can be used in the process of minimization of interrupt polling as disclosed in Ser. No. 07/151,123, filed Feb. 1, 1988 for A SYSTEM AND METHOD FOR SIMULATING THE I/O OF A PROCESSING SYSTEM, currently co-pending, and assigned to the same assignee as the present invention, which is hereby incorporated by reference.

II. Instruction Address Translation

The simulator in the preferred embodiment of this invention translates the instructions of Intel's iAPX 0286 processor (first processor) used in the IBM PC AT into simulator instructions of the ROMP processor (second processor) used in the IBM RT PC. These translations are saved for reuse when the application program originally written for the simulated first processor transfers control to that same address again. For instructions that do not transfer control, determining the next instruction pointer (IP) is accomplished by adding the length of the instruction to the instruction's instruction pointer (IP). A similar sequence of sequential simulator instructions are generated by the simulator. Because the instructions follow sequentially, locating the corresponding translations is not required.

Another class of instructions transfers control statically. That is, the new instruction pointer is computed by adding or subtracting a fixed displacement from the instruction's pointer. These are known as relative branches. The simulator generates second processor relative branches to the corresponding translation.

The instruction set contains three instructions that transfer control within a code segment to a new instruction pointer which cannot be determined statically since it is loaded from a register or memory at run time. The RETurn from subroutine instruction is one of these. The speed with which this instruction is simulated affects the overall performance of the simulator, especially if a processing system spends more time executing a RETurn instruction than any other instruction. The other two instructions are the JuMP indirect and the CALL indirect (register or memory). They will be treated the same as the RETurn instruction.

For example, a first processor addresses instructions by using two registers. The code segment register **33**, FIG. 3B describes the location of a 64K block **119** of memory **120**, FIG. 3C. The instruction pointer register **31**, FIG. 3B, is an offset into that code segment **33**. The instruction pointer **31** describes at which of the 64K bytes the instruction **100** is located. The first processor addresses instructions by indexing into a code segment **33** with the instruction pointer **31**, FIG. 3C.

The simulator of this invention uses the data structures shown in FIG. **6** to map first processor instruction addresses **100** which consist of a code segment **33** and an instruction pointer **31** to the address in memory **120** of the corresponding second processor where a sequence of second processor instructions **130** perform the same function.

In the method of this invention, a new instruction pointer **31**, and code segment **33**, whose values are determined at run time, are converted into the simulator machine (second processor) address of the corresponding translation for those instructions that transfer control dynamically, such as the RETurn from subroutine instruction, JuMP indirect, and the CALL indirect, or software interrupt instructions.

The simulator uses a three tier approach (FIG. **7**) to simulate these three instructions that transfer control dynamically. The three tier approach is organized in succession. The fastest and most likely case is performed first. The second approach is used should the first approach fail. The third approach is the slowest approach which guarantees a successful conversion of the instruction pointer to an address of the simulator machine. FIG. **6** illustrates the second and third approaches.

Referring to FIGS. **5** and **7**, the first operation performed on the new instruction pointer is to compare it with the value produced by the previous execution of that instruction, step **141**. If the values match, a simulator machine relative branch is performed to the corresponding address. An exclusive OR operation determines if the values match, step **142**, and a conditional branch transfers control if they do, step **143**. This allows for a fast computation of the look up address.

Referring to FIGS. **6** and **7**, should the above approach fail, i.e, the instruction pointer **31** is different from the previous time that particular instruction had executed, a table look up is performed using the address mapping translate look-aside buffer **34** which is similar to a hardware translate look aside buffer. The conversion from a first processor instruction to a second processor instruction uses the method described below.

Second processor instructions **130**, when translated, assume certain values of first processor registers, called attributes. These attributes can be used by the translator to generate more efficient code in certain cases. For example, if the stack alignment is even, half word instructions can be used to transfer data to and from the stack. Otherwise, two separate byte instructions must be used. The value of the code segment **33** and the alignment of the stack pointer 35 for a block of first processor instructions 100 are known as the attributes of the block. Both the code segment 33 and the stack pointer 35 are 16 bit fields. The attributes occur in the code block header 36, and the address mapping translate look-aside buffer 34. Translations 130 of first processor instructions 100 with different attributes are kept separate.

The method takes the lower thirteen bits 32 of the instruction pointer 31 and uses this as an index into the table 34 which is aligned at a 64K byte boundary at a fixed virtual address, step 144. The entry contains two words. The first word contains the attributes. The first 16 bits CS1 45 are the value of the code segment 33, the next bit 47 contains the alignment of the stack pointer 35 denoted as S1, the next bit 46 denoted as V1 is a valid bit which is zero if the entry is not valid, and one if it is a valid entry. There are some unused bits 51. The last 3 bits 48 of the 32 bit word are the upper three bits 49 of the instruction pointer 31 denoted as IP1.

Therefore, the method is to index the address mapping table 34 with the lower 13 bits 32 of the instruction pointer 31, step 144, and compare the first 16 bits CSl 45 with the current value of the code segment 33, step 145. This indicates that the instructions are in the same code segment indicating that the previous instruction may have been executed recently. If that matches, the lower bit 41 of the stack pointer 35 is compared with S1 47, to insure that the assumptions made in the translations about the alignment of the stack are not violated, step 146. If this matches, and V1 46 is on indicating a valid entry, and IP1 48 matches the upper 3 bits 49 of the instruction pointer 31, then there is a hit in the addressing mapping translate look-aside table. That is, the current instruction is exactly identified with an earlier translated second processor instruction. The next word 52 is a 32 bit address of the second processor instructions 130 that simulate the first processor instruction 100, step 175. If there is a miss, i.e., anyone of the above comparisons fails, translation proceeds by accessing the hash table 37 as described in the third approach below, and the address mapping translate look-aside buffer 34 entry is updated with the new attributes and the new branch address for future reference.

In the event that the look up is successful, the XIL and BEQ instructions are modified accordingly. The value to be EXCLUSIVE ORed by XIL will contain the new instruction pointer, and the relative offset for the branch instruction will indicate the new target address. Control is transferred to the new target.

The third approach also is shown in FIGS. 6 and 7. The middle six bits 38 of the code segment 33 are XOR'ed with the concatenation of the lower file bits 39 of the code segment 33 and the lower bit 41 of the stack pointer 35. That gives a six bit index into the code block hash table 37, step 148. The code block hash table has 64 entries. Each entry 53 points either to a null which indicates there is no entry, and implies a new translation is needed, step 174, or contains a pointer to a code block header 36.

Each code block header 36 describes the available second processor translations for a given attribute. The first field 55 in the code block header 36 contains a pointer to the next code block header 56 that hashed to that same entry 53 in the code block hash table 37. The next field 57 contains the attributes of the code block. The attributes include the code segment 33 in block 58 identified as CS2, and the alignment of the stack pointer 35 in block S2 59. For simplicity, the valid bit V1 46 is repeated as V2 60. The code block headers 36 are searched for the code block header 36 that has the same attributes as the instruction executing at run time, step 149. The next two fields 62, 63, contain the minimum and maximum first processor addresses for which there are translations with these attributes. The next field 64 is a pointer to the root of a tree that describes all the code blocks 83 for which there are translations with the specific attribute.

Each of the nodes of the tree, i.e. code block 83, contains minimum and maximum instruction pointers 65, 66 for which there is a second processor translation 130 within a range of first processor instruction pointers 31. There is a pointer to the left and right son 67, 68, respectively. The left son 67 points to a subtree 69 with lower or equal minimum instruction pointers 31. The right son 68 points to a subtree 70 with higher minimum instruction pointers 31. The nodes of the tree are searched to find the subtree having a range of instruction addresses within which the current instruction address falls, step 151. The next field 71 is a pointer to the end 73 of the code block 83 of second processor translations 130.

There is an array 44 with as many entries 72 as there are in the range of instruction pointers 31, i.e. the lower bound of the array 44 is the minimum instruction pointer, and the maximum bound of the array 44 is the maximum instruction pointer. Each of these entries in array 44 contain either 0 which indicates a first processor instruction 100 for which there is no second processor translation, or a pointer if it corresponds to a valid first processor instruction entry point. Each entry in the array 44 is a half-word. If there is a valid entry, the entry contains the offset from the beginning of the array 44 to the appropriate entry point for that particular instruction pointer, step 173.

In summary, the table look up fails in the unlikely event that two instruction pointers hash to the same entry in the table, or if this is the first time that the application program is transferring control to that instruction pointer. In either case, the simulator will access slower data structures, i.e., the binary tree 83, to convert the address. If not found, the translator 27 is invoked to generate simulator machine (second processor) equivalent instructions for the new block of first processor instructions, step 174, and the translate look-aside buffer 34, and the code block 82 are updated.

III. Memory Mapping

Referring to FIG. 8, the memory of a processing system 1 can be classified by type and contents. Region I is random access memory (RAM) 152. The contents of RAM 152 are instructions and data. The CPU 93 can read and write to this region of memory.

The second region of memory is called adapter memory (video) 153. The CPU can use explicit IN/OUT instructions to directly access the I/O adapter 91 connected to an output device 92, such as a display, shown as line 167, or the CPU 93 can use memory instructions to access the I/O adapter 91 through a video buffer 153. Using memory instructions to a video buffer 153 for output to a device 92 is referred to as memory mapped I/O since it really is I/O going to an adapter, although through a memory location. Memory mapped I/O allows a wider range of instructions to be used since the CPU has more instructions that go to memory than it has to perform explicit I/O (IN/OUT instructions). The contents of the video range of memory is output data, i.e. memory mapped I/O.

The third type of memory is region III called read only storage (ROS) **154**, otherwise referred to as read only memory (ROM). The contents of ROS **154** is mostly instructions, although there could be data also. In either case, the contents of ROS is never modified.

The address space **150** of a processing system such as the IBM PC AT divides logically into these three areas. The first is a 640K byte region of processor Read/Write storage, i.e. RAM **152**. The second is a 256K byte region reserved for I/O adapters, i.e. video **153**. These include data buffers for devices and device dependent ROS. The third is a 128K byte region of processor ROS **154**. This ROS contains BIOS and Basic. There is memory above the 1 meg range **166** that follows the BIOS area **155**. Since, the simulator of this preferred embodiment does not support the **286** protected mode, this area **176** of memory **150** is not provided for.

In order to translate the addresses of a first processing system into the addresses of a second processing system, the memory of the first processing system must be mapped into the memory of the second processing system.

To map memory, two shared memory segments **190**, **180** (FIG. **9**) of the operating system of the second processing system are used. The first shared memory segment **190** is used to store an image **150** of the memory of the first processing system. The second shared memory segment **180** indicates for each memory location the type of contents contained in that image.

In the second processing system memory **190**, there is the 640K bytes of memory **152** that would reside in the first processing system, followed by the video range **153**, followed by an area **154** for ROS (read only storage), followed by the area **155** for BIOS. In front of the first memory segment **152**, the BIOS area **155** is replicated. This was done to simplify the mapping in the case where an application could address the BIOS **155** area of memory **150** such that the offset would wrap around to RAM **152**.

The first processing system truncates addresses to 20 bits. Therefore, if when generating an address the result is greater than 1 megabyte (20 bits), the hardware subtracts 1 megabyte from the address and in effect addresses the beginning of memory. To simulate this efficiently, when the first processing system addresses BIOS memory **155**, these addresses are mapped to the first company of **155** in segment 0×9, **190**. When an address exceeds the bounds of **155**, it is simply mapped over to memory area **152** as it would have on the first processing system. Therefore, the hexadecimal addresses **172–175** of these areas are offset into the shared memory segment **190**, referred to as segment 0×9, by 64K from the address of the areas in memory **150** of the first processing system.

As a result, a virtual memory segment in the processing system running the simulator is dedicated to contain an image of all PC AT processor storage consisting of region one **152**, and two copies of BIOS region **155**. Region two **153** will not be present when the second processing system has the output devices from the first processing system attached to it. If the output devices of the first processing system are not attached, an image of region two **153** will also exist.

The address locations **172–175** in memory segment **190** are the location of the actual first processing system memory image. This area **172–175** would look like the architecture of the first processing system being simulated and the memory available.

In addition to the memory image **172–175**, the translated code **130** is stored in area **157**. After the translated instructions are generated by the graph analysis **30** (FIGS. 3A, 3B, 3C, 3D) examining the instructions **100** of the first processing system that is being simulated, the translated instructions **130** are stored in area **157**.

The next area **158** of the shared memory segment **190** is an inter process communications area and run time environment area where routines that are called at run time are stored.

In addition to the virtual shared memory segment **190** referred to as segment 0×9, the processing system running the simulator reserves a special segment for I/O bus memory referred to as segment 0×F, **181**. Data is written into segment 0×F of the second processing system if the second processing system has the output device of the first processing system attached to it. An address location of the first processing system that falls within region II, **153**, will have a corresponding memory location in segment 0×9, **190**, or to segment 0×F, **181**, of the second processing system.

The system and method of the simulator of this invention utilizes a relocate table **195** to map the memory **150** of the first processing system into either segment 0×9, **190**, or segment 0×F, **181** of the second processing system. A first processing system such as the PC AT addresses memory locations by two components, a segment and an offset. In the PC AT, the segment is a 16 bit value that points to a 64K block of memory. The offset is also 16 bits, and gives the displacement within the segment. The address of the segment is computed by multiplying the segment value by 16.

The simulator utilizes a table **195** with **16** entries **201–216**, each entry having 32 bits, to map the memory address of a PC AT into the 32 bit memory address of a IBM RT PC. The upper 4 bits of a segment of a PC AT memory address identifies one of the **16** entries of the table. The simulator separates the memory addressing computation into two parts. First, at the time that the segment register is loaded, the simulator multiplies the segment by 16 and adds it to the entry in the table specified by the upper 4 bits of the segment. Second, at the time an instruction reads or writes to memory, the 16 bit offset is added to the 32 bit value computed in the previous step. This is the address used to access memory of the second processing system.

At simulator start up time, the relocate table **195** is initialized. The first **10** entries **201–210** correspond to RAM **152** and are initialized to 0×90010000 to point to segment 0×9 of the second processing system. Note that this initialization value includes a 64K offset that allows room for the first instance of BIOS **155** mapped to segment 0×9, **190**, preceding RAM **152**. The last entry **216**, corresponding to the segment that addresses BIOS **155**, is initialized with 0×90000000–0×F0000, or 0×8FF10000. In this fashion when the segment multiplied by 16 is added, the resulting value will be 0×90000000 which maps the BIOS **155** at the beginning of segment 0×9, **190**. The eleventh through fifteenth entries **211–215** are initialized with 0×F4000000 or 0×90010000 to point to either segment 0×F, **181** or segment 0×9, **190** depending on whether the corresponding output device of the first processing system is attached.

The status control segment, segment 0×8, **180** is stored in another shared memory segment of the operating system. The status control segment **180** keeps track of the type of contents of memory of the first processing system for segment 0×9, **190**. This information is used by the simulator to determine whether the memory location contains data, instruction code, or is a video entry.

The above has described how the simulator computes the address that should be used when simulating an instruction that reads or writes to memory. In the case of writing to memory, special actions may be required.

If an instruction is being stored into memory, then the simulator makes note whether the instruction has been translated. A check for instruction modification is performed to insure that the translated code is always correct. If an application performs instruction modification, the translated code for the original code instruction is purged, and the new instruction is translated to a new sequence of instructions of the simulator's processor. Other steps could be taken to guarantee that the correct translation of the instruction will be executed.

Likewise, the check for video updates is necessary to determine if the output to the output device needs to be further processed by the simulator. This would occur in the case where the output device of the first processing system is not attached to the second processing system running the simulator, and the output device of the first processing system therefore must be simulated. In the case of memory mapped I/O, it must be detected at the time of a store whether the special hardware that is representing the output data is being modified.

At the time of a store to memory, segment 0×8, **180** is used to determine whether a special action is required on either an instruction or memory mapped I/O. Segment 0×8, **180** has a byte to byte correspondence with segment register 0×9, **190**. Each byte in segment 0×8 indicates the type of contents of the corresponding byte of segment 0×9, **190**. A byte in segment 0×8, **180** will have a zero if the corresponding byte in segment 0×9, **190** is computational data. The byte in segment 0×8, **180** will have either a 1,2,4 or 6, if the corresponding byte in segment 0×9 contains an instruction. Every byte of segment 0×8, **180** which corresponds to the video range **153** of segment 0×9 will contain a **16** if an output device of the first processing system is not attached. The value of **16** will be loaded into segment 0×8, **180** at simulator start up time during configuration.

If it is determined during configuration that an output device of a first processing system is attached to the second processing system, the corresponding bytes of segment 0×8, **180** will be zero. This indicates that the relocate table **195**, which also had entries **211-215** initialized during configuration, will map output data from the first processing system to segment 0×F, **181**, which requires no further action by the simulator since an output device of the first processing system is attached.

Regardless of the location of the store to memory, either segment 0×9 or segment 0×F, the 32 bit value of the address is ANDed with 0×80FFFFFF which results in the corresponding address of segment 0×8, **180**.

The instruction sequence shown in FIG. **10** is used on each translation of a first processing system instruction which is able to modify memory. It should be noted that the segment registers were chosen so that the address of the status byte could be calculated by ANDing the address of the memory byte with 0×80FFFFFF. As a result, both segments 0×F and 0×9 are mapped to segment 0×8. When these four instructions of FIG. **10** are overlapped with the instructions used to simulate the first processing system instruction, only 5 cycles are required to check the memory update to see if it requires special processing. However, it may be necessary to further determine whether the address of the store was to segment 0×F to determine whether special action is required after the store to memory.

By keeping a status byte corresponding to each byte in the memory image **190** or the memory mapped I/O **181**, video updates, memory mapped I/O, and instruction modification, can be detected. A flag in the status segment 8, **180**, is kept to indicate the content type of memory as follows:

0=data
1=instruction entry point
2=subsequent byte of an instruction
4=merged instruction (first byte not valid entry point)
8=breakpoint set on this instruction
16=video

As shown above, a non zero value may indicate that further action may be required by the simulator. A "1" indicates that it is an entry point, and there is a translation of the first processing system instruction in the binary tree. A "2" means it is a subsequent which means that there is a first processor instruction being simulated or translated that is more than one byte long. This byte then corresponds to a following byte. This allows for the fact that it may take more than one byte to represent a first processor instruction. A flag of "4" indicates that there is a merge. A merge means that as a result of the graph analysis, it was determined that it would take less translated simulator instructions to simulate several combined first processor instructions than it would take to translate each first processor instruction separately. For example, the PUSH instruction **105** and the POP instruction **106** of FIG. **3A** have been merged. As a result, the value in the CS register has been moved into the DS register. Since the graph analysis determines that this is all that these two instructions **105**, **106**, are doing, the two instructions can be combined into one, and executed faster than executing two separate instructions.

A flag of "8" indicates that a breakpoint is set. This allows debuggers to work on the simulator. A flag of **16** indicates that the information is video data. This is how the simulator detects that an application has gone out and updated the video screen.

The above method increases performance over previous simulators that required branching to a subroutine which performed an extensive, time consuming check, processed the memory update, and then returned. Just a branch to a subroutine typically requires at least 5 cycles. This is exemplified since storing to memory is a very frequent operation. Therefore, any reduction in the overhead of a store to memory greatly increases the efficiency of the simulator.

Although the foregoing invention has been particularly shown and described with reference to the preferred embodiments thereof, it will be understood by those skilled in the art that other changes in form may be made without departing from the scope of the claims.

1 PC AT is a registered trademark of IBM Corporation in the United States
2 Personal System/2 is a registered trademark of IBM Corporation the United States
3 RT PC is a registered trademark of IBM Corporation in the United States
4 AIX is a trademark of the IBM Corporation in the United States

We claim:

1. A method for reducing the number of translated instructions required to simulate an instruction flow of control of an application running on a processing system, said method comprising the steps of:

converting, at run time, an instruction pointer of a next instruction of a first processor resulting from a dynamic control instruction to an address for a second processor translation of said first processor's next instruction; and

utilizing a succession of steps for said converting, said succession of steps using a single second processor instruction in a first successive step, a translate look-aside buffer having a limited number of first processor instruction pointers and corresponding second processor translation address in a second successive step, and a data structure having more of said second processor translation addresses than said look-aside buffer in a third successive step.

2. A method of simulating a first processor having a first instruction set having a a plurality of first instructions by a second processing system having a second processor having a second instruction set for running an application targeted for said first instruction set, said second processing system method comprising:

comparing, using a single second processor instruction, at run time a new instruction pointer of a next instruction of said first processor with a previous instruction pointer of a previous dynamic control instruction translated for said second processor, said previous instruction pointer being produced by a previous executing of said previous dynamic control instruction translation to determine a validity of an address for a second processor translation for said first processor's next instruction; and

branching to the address of the second processor translation of the next instruction if said run time new instruction pointer matches said previous instruction pointer.

3. A method of simulating a first processor having a first instruction set having a plurality of first instructions by a second processing system having a second processor having a second instruction set for running an application targeted for said first instruction set, said second processing system method comprising:

comparing, using a single second processor instruction, at run time a new instruction pointer of a next instruction of said first processor with a previous instruction pointer of a previous dynamic control instruction translated for said second processor, said previous instruction pointer being produced by a previous executing of said previous dynamic control instruction translation to determine a validity of an address for a second processor translation for said first processor's next instruction; and

utilizing a translate look-aside buffer, having first processor instruction pointers fewer in number than said plurality of first instructions, and corresponding second processor translation addresses, to determine the address for said second processor translation of said first processor's next instruction if said comparison fails.

4. A method of simulating a first processor having a first instruction set having a plurality of first instruction by a second processing system having a second processor having a second instruction set for running an application targeted for said first instruction set, said second processing system method comprising:

comparing, using a single second processor instruction, at run time a new instruction pointer of a next instruction of said first processor with a previous instruction pointer of a previous dynamic control instruction translated for said second processor, said previous instruction pointer being produced by a previous execution of said previous dynamic control instruction translation to determine a validity of an address for a second processor translation for said first processor's next instruction;

utilizing a translate look-aside buffer, having first processor instruction pointers fewer in number than said plurality of first instructions, and corresponding second processor translation addresses, to determine the address for said second processor translation of said first processor's next instruction if said comparison fails, and

searching a data structure, having more of said second processor translation addresses than said look-aside buffer, for determining said address of said next instruction if said address is not found with said translate look-aside buffer.

5. A method of simulating a first processor having a first instruction set having a plurality of first instructions by a second processing system having a second processor having a second instruction set for running an application targeted for said first instruction set, said second processing system method comprising:

comparing, using a single second processor instruction, at run time, a new instruction pointer of a next instruction of said first processor with a previous instruction pointer of a previous dynamic control instruction translated for said second processor, said previous instruction pointer being produced by a previous executing of said previous dynamic control instruction translation to determine a validity of an address for a second processor translation for said first processor's next instruction when the next dynamic control instruction returns to a same address of a last time the previous dynamic control instruction was executed;

using, if said new instruction pointer is not the same as said previous instruction pointer, a translate look-aside buffer, having a limited number of first processor instruction pointers and corresponding second processor translation addresses, to determine the address for said second processor translation of said first processor's next instruction when the dynamic control instruction returns to the same address that was returned to at least once before other then the last time; and

using, if said address is not found with said translate look-aside buffer, a data structure, having more of said second processor translation addressed than said look-aside buffer, to determine the address of the next instruction when the dynamic control instruction returns to a new address that has not be previously returned to.

6. A system having means for simulating a first processor having a first instruction set having a plurality of first instructions by a second processing system having a second processor having a second instruction set for running an application targeted for said first instruction set, said system comprising:

means for comparing, using a single second processor instruction, at run time a new instruction pointer of a next instruction of said first processor with a previous instruction pointer of a previous dynamic control instruction translated for said second processor, said previous instruction pointer being produced by a previous execution of said previous dynamic control instruction translation to determine a validity of an address for a second processor translation for said first processor's next instruction;

means for utilizing a translate look-aside buffer, having first processor instruction pointers fewer in number than said plurality of first instructions, and corresponding second processor translation addresses, to determine the address for said second processor translation of said first processor's next instruction if said comparison fails; and

means for searching a data structure, having more of said second processor translation addresses than said look-aside buffer, for determining said address of said next instruction is said address is not found with said translate look-aside buffer.

7. A computer program having means for simulating a first processor having a first instruction set having a plurality of first instructions by a second processing system having a second processor having a second instruction set for running an application targeted for said first instruction set, said computer program comprising:

means for comparing, using a single second processor instruction, at run time a new instruction pointer of a next instruction said first processor with a previous instruction pointer of a previous dynamic control instruction translated for said second processor, said previous instruction pointer being produced by a previous execution of said previous dynamic control instruction translation to determine a validity of an address for a second processor translation for said first processor's next instruction;

means for utilizing a translate look-aside buffer, having first processor instruction pointers fewer in number than said plurality of first instructions, and corresponding second processor translation addresses, to determine the address for said second processor translation of said first processor's next instruction if said comparison fails; and

means for searching a data structure, having more of said second processor translation addresses than said look-aside buffer, for determining said address of said next instruction if said address is not found with said translate look-aside buffer.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 5,167,023

DATED : Nov. 24, 1992

INVENTOR(S) : Arturo M. de Nicolas; John C. O'Quinn, III

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Col. 6, line 14, please delete "starting," and insert --starting--;

Col. 11, line 47, please delete "0286" and insert --80286--;

Col. 19, line 31, please delete "executing" and insert --execution--;

line 51, please delete "executing" and insert --execution--;

line 64, please delete "instruction" second occurrence and insert --instructions--;

Col. 20, line 35, please delete "executing" and insert --execution--;

line 54, please delete "addressed" and insert --addresses--;

line 57, please delete "be" and insert --been--;

Col. 21, line 20, please delete "is" first occurrence and insert --if--; and

Col. 22, line 5, after "instruction" please insert --of--.

Signed and Sealed this

Thirty-first Day of October 1995

Attest:

Bruce Lehman

BRUCE LEHMAN

Attesting Officer *Commissioner of Patents and Trademarks*