(19) **Bundesrepublik Deutschland**
Deutsches Patent- und Markenamt

(10) **DE 602 03 612 T2** 2006.03.09

(12) # Übersetzung der europäischen Patentschrift

(97) **EP 1 390 841 B1**
(21) Deutsches Aktenzeichen: **602 03 612.7**
(86) PCT-Aktenzeichen: **PCT/GB02/00688**
(96) Europäisches Aktenzeichen: **02 711 108.7**
(87) PCT-Veröffentlichungs-Nr.: **WO 02/097613**
(86) PCT-Anmeldetag: **18.02.2002**
(87) Veröffentlichungstag der PCT-Anmeldung: **05.12.2002**
(97) Erstveröffentlichung durch das EPA: **25.02.2004**
(97) Veröffentlichungstag der Patenterteilung beim EPA: **06.04.2005**
(47) Veröffentlichungstag im Patentblatt: **09.03.2006**

(51) Int Cl.[8]: ***G06F 9/318*** (2006.01)
G06F 9/455 (2006.01)

(30) Unionspriorität:
0113199 **31.05.2001** **GB**

(84) Benannte Vertragsstaaten:
DE, FR, GB, IT, NL

(73) Patentinhaber:
ARM Ltd., Cherry Hinton, Cambridge, GB

(74) Vertreter:
Dr. Weber, Dipl.-Phys. Seiffert, Dr. Lieke, 65183 Wiesbaden

(72) Erfinder:
SEAL, James, David, Cambridge CB5 8PR, GB;
NEVILL, Colles, Edward, Hemingford Grey, Huntingdon PE18 9DR, GB

(54) Bezeichnung: **DATENVERARBEITUNG MIT MEHRFACHBEFEHLSSÄTZEN**

Beschreibung

[0001] Die vorliegende Erfindung bezieht sich auf das Gebiet der Datenverarbeitungssysteme. Genauer gesagt bezieht sich diese Erfindung auf Datenverarbeitungssysteme, die in der Lage sind, Befehle von mehr als einem Befehlssatz auszuführen.

[0002] Es ist bekannt, Datenverarbeitungssysteme bereitzustellen, die in der Lage sind, Befehle von mehr als einem Befehlssatz auszuführen. Ein Beispiel solcher Systeme sind Prozessoren, die von ARM Limited aus Cambridge, England, hergestellt werden, die sowohl den 32-Bit-ARM-Befehlssatz als auch den 16-Bit-Thumb-Befehlssatz ausführen können.

[0003] Ein in jüngerer Zeit entwickelter Typ von Datenverarbeitungssystemen, die mehr als einen Befehlssatz ausführen, ist derjenige, welcher sowohl seinen maschineneigenen Befehlssatz als auch Java-Bytecode-Befehle auszuführen versucht (siehe WO-A-99/18486). Genauer gesagt wurden Java-Beschleunigungstechniken vorgeschlagen und entwickelt, die Spezial-Hardware zum Ausführen von Java-Bytecodes bereitstellen. Ein Beispiel eines solchen Ansatzes ist die Jazelle-Architekturerweiterung, die von ARM Limited aus Cambridge, England, entworfen wurde, die dazu dient, Java-Bytecodes mittels eines Prozessorkerns auszuführen, der auch maschineneigene ARM-Befehle ausführt.

[0004] Ein Problem bei der oben genannten Technik ist, daß einige Java-Bytecodes nicht gut geeignet sind, von einer bezüglich der Architektur relativ einfachen Hardwareerweiterung, die von Jazelle bereitgestellt wird, ausgeführt zu werden. Die komplexeren Java-Bytecodes werden dementsprechend an unterstützende Software zur Ausführung gegeben. Ein solcher Ansatz kann die Komplexität des in Hardware ausgeführten Java-Beschleunigungssystems niedrig halten, während volle Abdeckung aller Java-Bytecodes, die bei der Ausführung vorkommen und benötigt werden, geboten wird. Allerdings können es verschiedene Implementierungen der Java-Beschleunigungshardware, die entwickelt werden, um speziellen Umständen zu genügen und sich über die Zeit entwickeln, erforderlich machen, daß verschiedene Java-Bytecodes durch Softwareausführung unterstützt werden, anstatt von den bereitgestellten Hardwaremechanismen ausgeführt zu werden. Dies erfordert unvorteilhafterweise, daß für jede Version der Java-Beschleunigungshardware ein unterschiedlicher Satz von unterstützender Software entwickelt und getestet wird. Dies ist teuer und zeitaufwendig.

[0005] Ein weiteres Problem, das bei den bekannten Java-Beschleunigungstechniken entstehen kann, liegt darin, daß es relativ schwierig ist, Fehler zu suchen und das Arbeiten des Systems zu verfolgen, wenn es Java-Bytecodes ausführt. Insbesondere kann ein einzelner Java-Bytecode, der von der Hardware ausgeführt wird, in der Praxis eine große Anzahl von diskreten Verarbeitungsoperationen repräsentieren, die von dem Prozessorkern durchzuführen sind, und dennoch erlaubt die Art, in der Java-Bytecodes von den Beschleunigungshardware als atomar (unteilbar) behandelt werden, nicht das betriebsbereite Einfügen von Haltepunkten, schrittweise Verarbeiten oder andere nützliche Diagnosetechniken.

[0006] Unter einem Aspekt betrachtet stellt die vorliegende Erfindung eine Vorrichtung zum Verarbeiten von Daten unter der Steuerung von Programmanweisungen aus einem ersten Satz von Programmanweisungen oder Programmanweisungen aus einem zweiten Satz von Programmanweisungen bereit, wobei die Vorrichtung aufweist:
einen Software-Befehlsübersetzer, der so betreibbar ist, daß er eine Programmanweisung des zweiten Satzes von Programmanweisungen als eine Sequenz von Programmanweisungen des ersten Satzes von Programmanweisungen mit einem die Sequenz abschließenden Befehl zu übersetzen; und
einen Befehlsdecoder, der auf den die Sequenz abschließenden Befehl reagiert:

(i) wenn eine Befehlsausführungseinheit in Hardware für den zweiten Satz von Anweisungen nicht verfügbar ist, dann die Übersetzung der nächsten Programmanweisung des zweiten Satzes von Anweisungen mittels des Software-Befehlsübersetzers einleitet; und
(ii) wenn die Befehlsausführungseinheit in Hardware für den zweiten Satz von Anweisungen verfügbar ist, dann die Ausführung der nächsten Programmanweisung des zweiten Satzes von Anweisungen mittels der Befehlsausführungseinheit in Hardware einleitet.

[0007] Die Erfindung sieht eine neue Anweisung vor, die als ein eine Sequenz abschließender Befehl verwendet wird, wenn eine Anweisung eines zweiten Befehlssatzes (z. B. ein Java-Bytecode) als eine Sequenz von Anweisungen eines ersten Befehlssatzes (z. B. eine Sequenz von maschineneigenen Befehlen) ausgeführt wird. Der eine Sequenz abschließende Befehl reagiert auf die Anwesenheit oder das Fehlen einer verfügbaren Befehlsausführungseinheit in Hardware, indem er entweder die Verarbeitung der nächsten Anweisung des zweiten Befehlssatzes mittels der Befehlsausführungseinheit in Hardware einleitet oder mit dem Einsatz des Software-Befehlsübersetzers fortfährt. Somit kann ein Software-Befehlsübersetzer mit der Fähigkeit, alle Anweisungen des zweiten Befehlssatzes zu behandeln, vorgesehen und dennoch nur dann verwendet werden, wenn er benötigt wird, da der eine Sequenz abschließende Befehl standardmäßig eine nächste Anweisung an die Befehlsausführungseinheit in Hardware übergibt, wenn eine solche Befehlsausführungseinheit in

Hardware verfügbar ist (z. B. vorhanden und betriebsbereit). Wenn ein spezieller Ausführungsmechanismus in Hardware eine bestimmte Anweisung des zweiten Befehlssatzes nicht unterstützt, dann wird diese bei Bedarf an den Software-Befehlsübersetzer zur Ausführung weitergeleitet, aber wenn ein Ausführungsmechanismus in Hardware vorhanden ist, wird ein Versuch unternommen, einen Bytecode mittels dieser Hardware auszuführen, da dies, wenn es erfolgreich ist, viel schneller ist.

[0008] Man könnte meinen, daß das Vorsehen eines eine Sequenz abschließenden Befehls, der in der Lage ist zu arbeiten, um zu veranlassen, daß die nächste Anweisung des zweiten Befehlssatzes entweder einer Software-Übersetzung oder einer Hardwareausführung unterzogen wird, nachteilig wäre, da zusätzliches Einrichten und Verarbeiten innerhalb des Software-Übersetzer erforderlich wäre, um beide Aktionsmodi zu unterstützen und dennoch Hardwareausführung möglicherweise niemals stattfinden würde. Jedoch in der Praxis ist diese Art von Verarbeitungsaktionen, die nötig sind, um das Einleiten einer nachfolgenden Hardwareausführung vorzubereiten, entweder innerhalb der Software-Übersetzung bereits vorgenommen oder innerhalb von Verarbeitungszyklen, während deren etwa wegen Registersperre oder Ähnlichem im allgemeinen keine andere nützliche Verarbeitung möglich ist, ohnehin vorgesehen bzw. einfach bereitgestellt. Daher kann der eine Sequenz abschließende Befehl beide Arten von anschließenden Aktionen mit überraschend geringem Overhead innerhalb des Software-Befehlsübersetzer unterstützen.

[0009] Man sieht, daß das Erfassen eines aktiven Ausführers in Hardware eine Vielzahl von Formen annehmen kann. In bevorzugten Ausführungsformen verwendet dieses Erfassen mindestens ein Hardware-Befehlsausführungseinheits-Flag. Solche Flags sind relativ einfach unter Softwaresteuerung zu setzen und können Information liefern wie die, ob eine Befehlsausführungseinheit in Hardware vorhanden ist oder nicht und/oder ob eine Befehlsausführungseinheit in Hardware aktiv ist oder nicht.

[0010] Um die anschließende Verarbeitung einer nächsten auf den eine Sequenz abschließenden Befehl folgende Anweisung des zweiten Befehlssatzes zu vereinfachen und zu beschleunigen, gibt der eine Sequenz abschließende Befehl vorzugsweise eine Startadresse für die Softwareübersetzung zur Verwendung durch den Software-Befehlsübersetzer an, wenn dies der aufgerufene Mechanismus ist. Es hat sich herausgestellt, daß es innerhalb der Softwareübersetzung einer aktuellen Anweisung häufig Gelegenheiten gibt, einen Zeiger vorzubereiten, um die Übersetzung der nächsten Anweisung zu starten ohne ungebührlichen Einfluß auf die Ausführungsgeschwindigkeit der aktuellen Anweisung in einer Art und Weise, die die Gesamtverarbeitungsgeschwindigkeit verbessert.

[0011] Diese Startadresse für die Übersetzung der nächsten Anweisung kann an den Software-Befehlsübersetzer auf einer Vielzahl von Wegen übergeben werden. Die bevorzugten Mechanismen, dies zu erreichen, sind allerdings, die Adresse innerhalb eines Registers, das als ein Operand innerhalb des eine Sequenz abschließenden Befehls spezifiziert ist oder innerhalb eines vorher festgelegten Registers zu übergeben, das von einem eine Sequenz abschließenden Befehl immer zu diesem Zweck verwendet wird.

[0012] Ähnliche Mechanismen zum Bereitstellen einer Startadresse für die Hardware-Ausführung zur Verwendung durch die Befehlsausführungseinheit in Hardware können ebenso vorgesehen werden.

[0013] Eine bequeme, flexible und robuste Methode für das Bereitstellen des Software-Befehlsübersetzers ist eine Mehrzahl von Sequenzen von Programmanweisungen des ersten Satzes von Anweisungen, von denen jede eine entsprechende Anweisung innerhalb des zweiten Satzes von Anweisungen darstellt. Auf diese Codefragmente kann über eine Tabelle von Zeigern zugegriffen werden, die optional von den Programmanweisungen des zweiten Satzes von Anweisungen indiziert sein können. Eine Basisadresse der Tabelle von Zeigern kann innerhalb eines Basisadreßregisters bereitgestellt werden als ein bequemer Mechanismus, den Code des Software-Befehlsübersetzers vielseitiger darzustellen.

[0014] Während man sieht, daß die vorliegende Erfindung in einer breiten Vielfalt von Situationen anwendbar ist, erkennt man auch, daß sie besonders gut für Situationen geeignet ist, in denen der erste Satz von Anweisungen ein maschineneigener Befehlssatz eines Prozessorkerns ist, der zweite Satz von Anweisungen Java-Bytecode-Anweisungen sind und der eine Sequenz abschließende Befehl eine maschineneigene Anweisung des Prozessorkerns ist.

[0015] Unter einem anderen Aspekt betrachtet stellt die vorliegende Erfindung ein Verfahren zum Verarbeiten von Daten unter der Steuerung von Programmanweisungen aus einem ersten Satz von Programmanweisungen oder Programmanweisungen aus einem zweiten Satz von Programmanweisungen zur Verfügung, wobei das Verfahren die folgenden Schritte aufweist: Verwenden eines Software-Befehlsübersetzer zum Übersetzen einer Programmanweisung des zweiten Satzes von Programmanweisungen als eine Sequenz von Programmanweisungen des ersten Satzes von Programmanweisungen, die mit einem eine Sequenz abschließenden Befehl abschließt; und

als Reaktion auf den eine Sequenz abschließenden Befehl:

(i) wenn eine Befehlsausführungseinheit in Hardware für den zweiten Satz von Programmanweisungen nicht verfügbar ist, dann Einleiten der Übersetzung einer nächsten Programmanweisung des zweiten Satzes von Anweisungen mittels des Software-Befehlsübersetzers; und

(ii) wenn die Befehlsausführungseinheit in Hardware für den zweiten Satz von Programmanweisungen verfügbar ist, dann Einleiten der Ausführung der nächsten Programmanweisung des zweiten Satzes von Anweisungen mittels der Befehlsausführungseinheit in Hardware.

[0016] Unter einem weiteren Aspekt betrachtet stellt die vorliegende Erfindung ein Computerprogrammprodukt zur Steuerung einer Datenverarbeitungsvorrichtung zur Verfügung, um Daten unter der Steuerung von Programmanweisungen aus einem ersten Satz von Programmanweisungen oder Programmanweisungen aus einem zweiten Satz von Programmanweisungen zu verarbeiten, wobei das Computerprogrammprodukt aufweist:

eine Software-Befehlsübersetzerlogik, die betreibbar ist zum Übersetzen einer Programmanweisung des zweiten Satzes von Programmanweisungen als eine Sequenz von Programmanweisungen des ersten Satzes von Programmanweisungen, die mit einem eine Sequenz abschließenden Befehl abschließt, wobei der eine Sequenz abschließende Befehl dazu dient:

(i) wenn eine Befehlsausführungseinheit in Hardware für den zweiten Satz von Programmanweisungen nicht verfügbar ist, dann die Übersetzung einer nächsten Programmanweisung des zweiten Satzes von Anweisungen mittels der Software-Befehlsübersetzerlogik einzuleiten; und

(ii) wenn die Befehlsausführungseinheit in Hardware für den zweiten Satz von Anweisungen verfügbar ist, dann die Ausführung der nächsten Programmanweisung des zweiten Satzes von Anweisungen mittels der Befehlsausführungseinheit in Hardware einzuleiten.

[0017] Das Computerprogrammprodukt kann die Form des Unterstützungscodes zur Verwendung in Verbindung mit einem Hardwarebeschleuniger annehmen. Dieser Unterstützungscode kann zum Beispiel auf einem Datenspeicherungsmedium oder als Firmware innerhalb eines eingebetteten Verarbeitungssystems bereitgestellt werden oder dynamisch heruntergeladen werden, wenn dies erwünscht ist.

[0018] Ausführungsformen der Erfindung werden nun nur mittels eines Beispiels beschrieben unter Bezug auf die beigefügten Zeichnungen, von denen:

[0019] Fig. 1 ein Datenverarbeitungssystem darstellt, das eine Bytecode-Übersetzungshardware umfaßt;

[0020] Fig. 2 die Software-Befehlsübersetzung von Bytecodes schematisch darstellt;

[0021] Fig. 3 ein Flußdiagramm ist, das die Funktionsweise eines Codefragmentes innerhalb des Software-Befehlsübersetzers, das mit einem eine Sequenz abschließende Befehl endet, schematisch darstellt;

[0022] Fig. 4 ein Beispiel eines Codefragmentes ist, das anstelle eines Bytecodes ausgeführt wird;

[0023] Fig. 5 ein beispielhaftes Datenverarbeitungssystem darstellt, das über keine Bytecode-Ausführungsunterstützung in Hardware verfügt;

[0024] Fig. 6 ein Flußdiagramm ist, das die Aktion des Software-Befehlsübersetzers darstellt, wenn er mit dem System von Fig. 5 arbeitet;

[0025] Fig. 7 die Abbildung zwischen Java-Bytecodes und Verarbeitungsoperationen darstellt;

[0026] Fig. 8 eine programmierbare Übersetzungstabelle in der Form eines inhaltsadressierbaren Speichers darstellt;

[0027] Fig. 9 eine programmierbare Übersetzungstabelle in der Form eines wahlfrei zugreifbaren Speichers darstellt;

[0028] Fig. 10 ein Flußdiagramm ist, das die Initialisierung und Programmierung einer programmierbaren Übersetzungstabelle schematisch darstellt;

[0029] Fig. 11 ein Diagramm ist, das einen Teil der Verarbeitungspipeline innerhalb eines Systems schematisch darstellt, das Java-Bytecode-Übersetzung durchführt;

[0030] Fig. 12 eine Anweisung variabler Länge schematisch darstellt, die zwei Befehlswörter und zwei virtuelle Speicherseiten überspannt;

[0031] Fig. 13 einen Teil einer Pipeline eines Datenverarbeitungssystems schematisch darstellt, das einen Mechanismus zum Umgang mit Abbrüchen beim Vorab-Holen der in Fig. 12 dargestellten Art beinhaltet;

[0032] Fig. 14 einen logischen Ausdruck angibt, der eine Art ist zu spezifizieren, wie ein Abbruch beim Vorab-Holen der in Fig. 12 dargestellten Art erkannt werden kann;

[0033] Fig. 15 eine Anordnung von Unterstützungscode zur Abbruchbehandlung und Befehlsemulation schematisch darstellt;

[0034] Fig. 16 ein Flußdiagramm ist, das die Verarbeitung schematisch darstellt, die zum Umgang mit Abbrüchen beim Vorab-Holen von Bytecode-Anweisungen variabler Länge durchgeführt wird;

[0035] Fig. 17 die Beziehung zwischen einem Betriebssystem und verschiedenen von diesem Betriebssystem gesteuerten Prozessen darstellt;

[0036] Fig. 18 ein Verarbeitungssystem darstellt, das einen Prozessorkern und einen Java-Beschleuniger beinhaltet;

[0037] Fig. 19 ein Flußdiagramm ist, das die Funktionsweise eines Betriebssystems beim Steuern der Konfiguration eines Java-Beschleunigers schematisch darstellt;

[0038] Fig. 20 ein Flußdiagramm ist, das die Funktionsweise einer virtuellen Java-Maschine in Verbindung mit einem Java-Beschleunigungsmechanismus schematisch darstellt, die sie beim Steuern der Konfiguration des Java-Beschleunigungsmechanismus' verwendet;

[0039] Fig. 21 ein Datenverarbeitungssystem darstellt, das Bytecodeübersetzungshardware wie in Fig. 1 umfaßt, welches ferner ein Gleitkomma-Subsystem umfaßt;

[0040] Fig. 22 ein Datenverarbeitungssystem darstellt, das Bytecodeübersetzungshardware wie in Fig. 1 und ein Gleitkomma-Subsystem wie in Fig. 21 umfaßt, welches ferner ein Gleitkommaoperationsregister und ein Flag für einen Zustand "nicht abgehandelte Operation" umfaßt;

[0041] Fig. 23 die ARM-Gleitkomma-Anweisungen zeigt, die für Java-Gleitkomma-Anweisungen erzeugt werden;

[0042] Fig. 24 eine Sequenz von ARM-Anweisungen zeigt, die von der Java-Beschleunigungshardware für die 'dmul'- und 'dcmpg'-Anweisungen in Java erzeugt werden könnten;

[0043] Fig. 25 die Sequenz von Operationen zeigt, wenn eine 'dmul'-Anweisung gefolgt von einer 'dcmpg'-Anweisung ausgeführt wird, wobei eine nicht abgehandelte Gleitkommaoperation durch die Ausführung der FCMPD-Anweisung, die von der Java-Beschleunigungshardware für die 'dmul'-Anweisung in Java erzeugt wird, verursacht wird. Die abgebildete Sequenz von Operationen ist für ein System, das eine ungenaue Erkennung einer nicht abgehandelten Operation entsprechend Fig. 22 verwendet;

[0044] Fig. 26 den Zustand des Gleitkommaoperationsregisters und des Flag für den Zustand "nicht abgehandelte Operation" nach der Ausführung der FMULD-Operation in Fig. 25 zeigt;

[0045] Fig. 27 die Sequenz von Operationen zeigt, wenn eine 'dmul'-Anweisung gefolgt von einer 'dcmpg'-Anweisung ausgeführt wird, wobei eine nicht abgehandelte Gleitkommaoperation durch die Ausführung der FCMPD-Anweisung, die von der Java-Beschleunigungshardware für die 'dcmpg'-Anweisung in Java erzeugt wird, verursacht wird, wobei die abgebildete Sequenz von Operationen für ein System vorgesehen ist, das eine ungenaue Erkennung einer nicht abgehandelten Operation entsprechend Fig. 22 verwendet;

[0046] Fig. 28 den Zustand des Gleitkommaoperationsregisters und des Flag für den Zustand "nicht abgehandelte Operation" nach der Ausführung der FCMPD-Anweisung in Fig. 27 zeigt;

[0047] Fig. 29 die Sequenz von Operationen zeigt, wenn eine 'dmul'-Anweisung gefolgt von einer 'dcmpg'-Anweisung ausgeführt wird, wobei eine nicht abgehandelte Gleitkommaoperation durch die Ausführung der FMULD-Anweisung, die von der Java-Beschleunigungshardware für die 'dmul'-Anweisung in Java erzeugt wird, verursacht wird. Die abgebildete Sequenz von Operationen ist für ein System, das eine genaue Erkennung einer nicht abgehandelten Operation entsprechend Fig. 21 verwendet; und

[0048] Fig. 30 die Sequenz von Operationen zeigt, wenn eine 'dmul'-Anweisung gefolgt von einer 'dcmpg'-Anweisung ausgeführt wird, wobei eine nicht abgehandelte Gleitkommaoperation durch die Ausführung der FCMPD-Anweisung, die von der Java-Beschleunigungshardware für die 'dcmpg'-Anweisung in Java erzeugt wird, verursacht wird. Die abgebildete Sequenz von Operationen ist für ein System, das eine genaue Erkennung einer nicht abgehandelten Operation entsprechend Fig. 21 verwendet.

[0049] Fig. 1 stellt eine Datenverarbeitungssystem **2** dar, das einen Prozessorkern **4** wie einen ARM-Prozessor und Bytecode-Übersetzungshardware **6** (auch Jazelle genannt) umfaßt. Der Prozessorkern **4** beinhaltet eine Registerbank **8**, einen Befehlsdecoder **10** und einen Datenpfad **12** zum Durchführen unterschiedlicher Datenverarbeitungsoperationen auf Datenwerten, die innerhalb der Register der Registerbank **8** gespeichert sind. Ein Register **18** ist vorgesehen, das ein Flag **20** enthält, welches steuert, ob die Bytecode-Übersetzungshardware **6** gerade freigegeben oder gesperrt ist. Darüber hinaus steht ein Register **19** zur Verfügung, das ein Flag **21** enthält, welches anzeigt, ob die Bytecode-Übersetzungshardware gerade aktiv oder inaktiv ist. Mit anderen Worten zeigt das Flag **21** an, ob das Datenverarbeitungssystem gerade Java-Bytecodes oder ARM-Anweisungen ausführt. Es ist klar, daß in anderen Ausführungsformen die Register **18** und **19** ein

einzelnes Register sein könnten, das beide Flags **20** und **21** enthält.

[0050] Wenn Java-Bytecodes während des Betriebs ausgeführt werden und die Bytecode-Übersetzungshardware **6** aktiv ist, dann werden die Java-Bytecodes von der Bytecode-Übersetzungshardware **6** empfangen und dienen dazu, eine Sequenz von entsprechenden ARM-Anweisungen (in dieser speziellen, nicht-einschränkenden beispielhaften Ausführungsform) oder zumindest den Prozessorkern steuernde Signale zu erzeugen, die ARM-Anweisungen repräsentieren, die anschließend an den Prozessorkern **4** übergeben werden. Somit kann die Bytecode-Übersetzungshardware **6** einen einzelnen Java-Bytecode auf einen Sequenz von entsprechenden ARM-Anweisungen abbilden, die von dem Prozessorkern **4** ausgeführt werden. Wenn die Bytecode-Übersetzungshardware inaktiv ist, wird sie umgangen, und normale ARM-Anweisungen können an den ARM-Befehlsdecoder **10** geliefert werden, um den Prozessorkern **4** gemäß seines maschineneigenen Befehlssatzes zu steuern. Man erkennt durchweg, daß die Sequenzen von ARM-Anweisungen gleichermaßen Sequenzen von Thumb-Anweisungen und/oder Mischungen von Anweisungen aus unterschiedlichen Befehlssätzen sein könnten, und solche Alternativen werden angestrebt und sind hier umfasst.

[0051] Man sieht, daß die Bytecode-Übersetzungshardware **6** nur Hardware-Übersetzungsunterstützung für eine Teilmenge der möglichen Java-Bytecodes, die angetroffen werden können, bieten kann. Bestimmte Java-Bytecodes können eine solche ausgiebige und abstrakte Verarbeitung erfordern, daß es nicht effizient wäre zu versuchen, diese in Hardware auf entsprechende ARM-Anweisungs-Operationen abzubilden. Dementsprechend wird dann, wenn die Bytecode-Übersetzungshardware **6** einen solchen nicht in Hardware unterstützten Bytecode antrifft, sie einen Software-Befehlsübersetzer anstoßen, der in maschineneigenen ARM-Anweisungen geschrieben ist, um die Verarbeitung durchzuführen, die von diesem nicht in Hardware unterstützten Java-Bytecode spezifiziert ist.

[0052] Der Software-Befehlsübersetzer kann so geschrieben werden, daß er Software-Unterstützung für alle möglichen Java-Bytecodes zu bieten, die übersetzt werden können. Wenn die Bytecode-Übersetzungshardware **6** vorhanden und freigegeben ist, dann werden normalerweise nur diejenigen Java-Bytecodes, die nicht in Hardware unterstützt sind, an die relevanten Codefragmente innerhalb des Software-Befehlsübersetzers verwiesen. Sollte die Bytecode-Übersetzungshardware **6** jedoch nicht bereitstehen oder gesperrt sein (wie während einer Fehlersuche oder Ähnlichem), dann werden alle Java-Bytecodes an den Software-Befehlsübersetzer verwiesen.

[0053] Fig. 2 stellt die Aktion des Software-Befehlsübersetzers schematisch dar. Ein Strom von Java-Bytecodes **22** stellt ein Java-Programm dar. Zwischen diese Java-Bytecodes können Operanden eingefügt sein. Daher kann im Anschluß an die Ausführung eines gegebenen Java-Bytecodes der nächste auszuführende Java-Bytecode in der unmittelbar folgenden Byteposition erscheinen oder einige Bytepositionen später liegen, wenn dazwischenliegende Operandenbytes vorhanden sind.

[0054] Gemäß Fig. 2 wird ein Java-Bytecode BC4 angetroffen, der nicht von der Bytecode-Übersetzungshardware **6** unterstützt wird. Dies ruft eine Ausnahmebedingung innerhalb der Bytecode-Übersetzungshardware **6** hervor, die veranlaßt, daß innerhalb einer Tabelle von Zeigern **24** nachgesehen wird, wobei der Bytecode-Wert BC4 als ein Index verwendet wird, um einen Zeiger P#4 auf ein Codefragment **26** zu lesen, das die von dem nicht in Hardware unterstützten Bytecode BC4 spezifizierte Verarbeitung durchführt. Ein Basis-Adreßwert der Zeigertabelle kann auch in einem Register gespeichert sein. In das gewählte Codefragment wird dann eingetreten, wobei R14 auf den nicht unterstützen Bytecode BC4 zeigt.

[0055] Da es **256** mögliche Bytecode-Werte gibt, enthält die Zeigertabelle **24** wie dargestellt **256** Zeiger. In ähnlicher Weise werden **256** Codefragmente mit maschineneigenen ARM-Anweisungen bereitgestellt, um die von allen möglichen Java-Bytecodes spezifizierte Verarbeitung durchzuführen. (Diese können weniger als **256** sein in Fällen, in denen zwei Bytecodes dasselbe Codefragment verwenden können). Die Bytecode-Übersetzungshardware **6** wird typischerweise Hardwareunterstützung für viele der einfachen Java-Bytecodes bereitstellen, um die Verarbeitungsgeschwindigkeit zu steigern, und in diesem Fall werden die entsprechenden Codefragmente innerhalb des Software-Befehlsübersetzers nie verwendet, außer wenn dies erzwungen wird wie etwa während einer Fehlersuche oder unter anderen Umständen wie etwa Abbrüchen beim Vorab-Holen wie später diskutiert wird. Da diese jedoch typischerweise die einfacheren und kürzeren Codefragmente sind, zieht ihr Bereitstellen einen relativ geringen zusätzlichen Speicheroverhead nach sich. Außerdem wird dieser kleine zusätzliche Speicheroverhead durch das dann generische Wesen des Software-Befehlsübersetzers und seiner Fähigkeit, mit allen möglichen Java-Bytecodes umzugehen unter Umständen, in denen die Bytecode-Übersetzungshardware nicht vorhanden oder gesperrt ist, mehr als kompensiert.

[0056] Man sieht, daß jedes der Codefragmente **26** von Fig. 2 von einem eine Sequenz abschließenden

Befehl BXJ abgeschlossen wird. Die Aktion dieses eine Sequenz abschließenden Befehls BXJ variiert abhängig von dem Zustand des Datenverarbeitungssystems **2**, wie in Fig. 3 dargestellt wird. Fig. 3 ist ein Flußdiagramm, das in einer höchst schematischen Form die Verarbeitung darstellt, die von einem Codefragment **26** innerhalb des Software-Befehlsübersetzers durchgeführt wird. Bei Schritt **28** wird die von dem gerade übersetzten Java-Bytecode spezifizierte Operation durchgeführt. Bei Schritt **30** wird der nächste auszuführende Java-Bytecode aus dem Bytecodestrom **22** gelesen und der Bytecode-Zeiger innerhalb des Java-Bytecode-Stromes **22**, der diesem nächsten Java-Bytecode entspricht, wird innerhalb eines Registers der Registerbank **8**, nämlich R14, gespeichert. Daher ist für den Java-Bytecode BC4 von Fig. 2 BC5 der nächste Java-Bytecode, und Register R14 wird mit einem Zeiger auf die Speicherstelle des Java-Bytecodes BC5 geladen.

[0057] Bei Schritt **32** wird der Zeiger innerhalb der Zeigertabelle **24**, der dem nächsten Java-Bytecode BC5 entspricht, aus der Zeigertabelle **24** gelesen und in einem Register der Registerbank **8**, nämlich Register R12, gespeichert.

[0058] Man sieht, daß Fig. 3 die Schritte **28**, **30** und **32** als separat und sequentiell ausgeführt darstellt. Jedoch kann die Verarbeitung der Schritte **30** und **32** gemäß bekannten Programmiertechniken bequem in die Verarbeitung von Schritt **28** eingefügt werden, um von den ansonsten verschwendeten Verarbeitungsmöglichkeiten (Zyklen) innerhalb der Verarbeitung von Schritt **28** Gebrauch zu machen. Daher kann die Verarbeitung der Schritte **30** und **32** mit relativ geringem Overhead für die Ausführungsgeschwindigkeit bereitgestellt werden.

[0059] Schritt **34** führt den eine Sequenz abschließenden Befehl BXJ mit dem Register R14 als Operand angegeben aus.

[0060] Vor der Ausführung des BXJ-Befehls bei Schritt **34** wurde der Zustand des Systems eingerichtet, und zwar mit dem Zeiger auf den nächsten Java-Bytecode innerhalb des Java-Bytecode-Stromes **22**, der in dem Register R14 gespeichert ist und mit dem Zeiger auf das Codefragment, das diesem nächsten Java-Bytecode, der in Register R12 gespeichert wird. Die Wahl der speziellen Register könnte variiert werden und keines, eines oder beide als Operanden des die Sequenz abschließenden Befehls angegeben oder durch die Architektur vorab festgelegt und definiert werden.

[0061] Die Schritte **28**, **30**, **32** und **34** sind vorwiegend Software-Schritte. Die an Schritt **34** anschließenden Schritte in Fig. 3 sind vorwiegend Hardware-Schritte und finden ohne separate, identifizierbare Programmanweisungen statt. Bei Schritt **36** erkennt die Hardware, ob die Bytecode-Übersetzungshardware **6** aktiv ist oder nicht. Sie tut das, indem sie die Werte der Registerflags für das Vorhandensein und das Freigegeben-Sein der Bytecode-Übersetzungshardware **6** liest. Andere Mechanismen zum Erkennen des Vorhandenseins von aktiver Bytecode-Übersetzungshardware **6** sind auch möglich.

[0062] Wenn die Bytecode-Übersetzungshardware **6** vorhanden und freigegeben ist, dann geht die Verarbeitung weiter zu Schritt **38**, bei dem die Kontrolle an die Bytecode-Übersetzungshardware **6** zusammen mit dem Inhalt des Registers R14 übergeben wird, das den Bytecodezeiger auf einen Bytecode innerhalb des Bytecode-Stroms **22** angibt, den die Bytecode-Übersetzungshardware **6** als seinen nächsten Bytecode auszuführen versuchen sollte. Dann endet die Aktion des dargestellten Codefragments **26**.

[0063] Wenn alternativ bei Schritt **36** festgestellt wird, daß es keine Bytecode-Übersetzungshardware **6** gibt oder die Bytecode-Übersetzungshardware gesperrt ist, dann geht die Verarbeitung zu Schritt **40** über, bei dem ein Sprung innerhalb des maschineneigenen ARM-Anweisungscodes vorgenommen wird, um die Ausführung des Codefragmentes innerhalb des Software-Befehlsübersetzers aufzunehmen, auf das von der in Register R12 gespeicherten Adresse gezeigt wird. Daher wird eine schnelle Ausführung des nächsten Codefragmentes eingeleitet, was einen Vorteil bei der Verarbeitungsgeschwindigkeit ergibt.

[0064] Fig. 4 stellt ein spezielles Codefragment genauer dar. Dieses spezielle Beispiel ist eine Java-Bytecode zur Ganzzahl-Addition, dessen Mnemonik iadd ist.

[0065] Die erste maschineneigene ARM-Anweisung verwendet den um Eins inkrementierten Bytecode-Zeiger in Register R14, um den nächsten Bytecode-Wert zu lesen (eine Ganzzahl-Addier-Anweisung hat keine nachfolgenden Bytecode-Operanden und somit folgt der nächste Bytecode unmittelbar dem aktuellen Bytecode). Der Bytecode-Zeiger in Register R14 wird auch mit dem inkrementierten Wert aktualisiert.

[0066] Die zweite und dritte Anweisung dienen dazu, vom Stack die beiden zu addierenden Ganzzahl-Operanden zu holen.

[0067] Die vierte Anweisung nutzt das, was sonst wegen der Registersperre auf Register R0 ein verschwendeter Verarbeitungszyklus wäre, um den Adreßwert des Codefragmentes für den nächsten in Register R4 gespeicherten Bytecode zu holen und diese Adresse in Register R12 zu speichern. Ein Register Rexc wird verwendet, um einen Basiszeiger

auf den Beginn der Zeigertabelle **24** zu speichern.

[0068] Die fünfte Anweisung führt die von dem Java-Bytecode spezifizierte Ganzzahl-Addition durch.

[0069] Die sechste Anweisung speichert das Ergebnis des Java-Bytecodes zurück auf den Stack.

[0070] Die letzte Anweisung ist der eine Sequenz abschließende Befehl BXJ, spezifiziert mit dem Operanden R12. Das Register R12 speichert die Adresse des ARM-Codefragments, das benötigt wird, um den nächsten Java-Bytecode in Software zu übersetzen, sollte Software-Übersetzung benötigt werden. Die Ausführung des BXJ-Befehls stellt fest, ob freigegebene Bytecode-Übersetzungshardware **6** vorhanden ist oder nicht. Wenn sie vorhanden ist, dann geht die Kontrolle an diese Bytecode-Übersetzungshardware **6** zusammen mit dem in Register R14 gespeicherten Operanden über, das die nächste Bytecode-Adresse spezifiziert. Wenn aktive Bytecode-Übersetzungshardware **6** nicht vorhanden ist, dann wird die Ausführung des Codefragmentes für den nächsten Bytecode, auf das von der Adresse in Register R12 gezeigt wird, gestartet.

[0071] Fig. 5 stellt ein Datenverarbeitungssystem **42** ähnlich dem von Fig. 1 schematisch dar, außer daß in diesem Fall keine Bytecode-Übersetzungshardware **6** vorgesehen ist. In diesem System zeigt Flag **21** immer an, daß ARM-Anweisungen ausgeführt werden und Versuche, in die Ausführung von Java-Bytecode mit einem BXJ-Befehl einzusteigen immer so behandelt werden, als ob die Bytecode-Übersetzungshardware **6** gesperrt wäre, wobei Flag **20** ignoriert wird.

[0072] Fig. 6 stellt ein Flußdiagramm der von dem System **42** beim Ausführen von Java-Bytecode durchgeführten Verarbeitung dar. Dies ist ähnlich der Verarbeitung von Fig. 3, bei der derselbe Software-Übersetzercode verwendet wird, außer daß in diesem Fall beim Ausführen des eine Sequenz abschließenden Befehls BXJ niemals die Möglichkeit einer Bytecode-Unterstützung in Hardware besteht und dementsprechend die Verarbeitung immer mit einem Sprung weitergeht, um das Codefragment auszuführen, auf das R12 als das Codefragment für den nächsten Java-Bytecode zeigt.

[0073] Man erkennt, daß der Software-Befehlsübersetzer in diesem Fall als maschineneigene ARM-Anweisungen bereitgestellt wird. Der Software-Befehlsübersetzer (und anderer Unterstützungscode) kann als ein separates eigenständiges Computerprogrammprodukt zur Verfügung gestellt werden. Dieses Computerprogrammprodukt kann über Aufnahmemedien wie eine Diskette oder eine CD verteilt werden oder könnte dynamisch über eine Netzverbindung heruntergeladen werden. Im Kontext von eingebetteten Verarbeitungsanwendungen, für die die vorliegende Erfindung besonders gut geeignet ist, kann der Software-Befehlsübersetzer als Firmware innerhalb eines nur lesbaren Speichers oder irgendeiner anderen nicht-flüchtigen Programmspeichereinrichtung innerhalb eines eingebetteten Systems zur Verfügung gestellt werden.

[0074] Fig. 7 stellt die Beziehung zwischen Java-Bytecodes und den Verarbeitungsoperationen, die sie spezifizieren, dar. Wie man aus Fig. 7 erkennt, sehen die 8-Bit-Java-Bytecodes **256** mögliche unterschiedliche Bytecode-Werte vor. Wie innerhalb der Java-Standards spezifiziert unterliegen die ersten **203** dieser Java-Bytecodes festen Bindungen zu entsprechenden Verarbeitungsoperationen wie etwa das zuvor diskutierte iadd. Die letzten beiden Bytecodes, nämlich **254** und **255**, sind in The Java Virtual Machine Specifikation als durch die Implementierung definiert angegeben. Daher steht es einer Java-Implementierung frei, diesen Bytecodes feste Bindungen zuzuweisen. Alternativ kann eine Java-Implementierung festlegen, diese als mit programmierbaren Bindungen versehen zu behandeln. Jazelle spezifiziert feste Bindungen für diese Bytecodes. Zwischen den Bytecode-Werten **203** und **253** einschließlich können programmierbare Bindungen wie von einem Benutzer gewünscht spezifiziert werden. Diese werden typischerweise verwendet, um Bindungen zwischen Bytecodes und Verarbeitungsoperationen bereitzustellen wie Quick-Form-Bytecode, die während der Laufzeit aufgelöst werden (siehe The Java Virtual Machine Specification, Autoren Tim Lindholm und Frank Yellin, Verlag Addison Wesley, ISBN 0-201-63452-X).

[0075] Aus Fig. 7 erkannt man, daß, während Hardware-beschleunigte Übersetzungstechniken gut geeignet sind, um mit festen Bindungen umzugehen, diese Techniken weniger gut geeignet sind, mit programmierbaren Bindungen umzugehen. Während es möglich wäre, alle programmierbaren Bindungen mittels Software-Übersetzungstechniken wie Übersetzen der relevanten Bytecodes als durch entsprechende Codefragmente repräsentiert zu behandeln, wäre dies langsam für das, was in einigen Fällen durchsatzkritische Bytecodes sein können.

[0076] Fig. 8 stellt eine Form einer programmierbaren Übersetzungstabelle dar. Diese programmierbare Übersetzungstabelle **100** liegt in der Form eines inhaltsadressierbaren Speichers vor. Ein zu übersetzender Bytecode wird in ein CAM-Nachschlage- bzw. -Lookup-Array **102** eingegeben (A.d.Ü.: content addressable memory, CAM). Wenn dieses Array **102** einen passenden Bytecode-Eintrag enthält, dann wird ein Treffer erzeugt, der veranlaßt, daß ein entsprechender, eine Operation spezifizierender Wert ausgegeben wird, d. h.
wenn es einen passenden Bytecode-Eintrag in der

CAM-Tabelle gibt, dann verwendet die Hardware den Operation-spezifizierenden Code, um eine in Hardware durchzuführende Operation zu ermitteln, führt diese Operation durch und geht zum nächsten Bytecode weiter; wenn es keinen passenden Bytecode-Eintrag in der CAM-Tabelle gibt, dann wird der Bytecode als nicht-hardwareunterstützt behandelt und sein Codefragment aufgerufen.

[0077] In diesem Beispiel sind die eine Operation spezifizierenden Werte 4-Bit-Werte, und der CAM-Eintrag, der den Treffer ergeben hat, entspricht dem Bytecode bc6. Wie man aus Fig. 7 versteht, haben alle Bytecodes, die Gegenstand solcher programmierbaren Übersetzung sein können, ihre beiden höchst-signifikanten Bits auf "1" und dementsprechend brauchen nur die niedrigsignifikanten 6 Bits des Bytecodes in das Array **102** eingegeben zu werden.

[0078] Die programmierbare Übersetzungstabelle **100** in diesem Beispiel hat acht Einträge. Die Anzahl von vorhandenen Einträgen kann abhängig von der Menge von Hardware-Ressourcen, die sich dieser Aufgaben widmen soll, variiert werden. In einigen Beispielen sind möglicherweise nur vier Einträge vorgesehen, während in anderen zehn Einträge angemessen sein können. Es kann auch möglich sein, einen Eintrag für jeden möglichen Bytecode mit programmierbarer Bindung vorzusehen.

[0079] Man erkennt, daß in dem Fall, daß die verfügbaren Ressourcen für programmierbare Abbildung zuerst mit den kritischsten Übersetzungen gefüllt werden, die weniger kritischen Übersetzungen dann Gegenstand von Software-Übersetzung sein können. Das Bereitstellen des Software-Übersetzers in Verbindung mit der programmierbaren Übersetzungstabelle ermöglicht es, daß die Konfiguration des Systems und die Programmierung der Tabelle vorgenommen wird, ohne daß es nötig ist zu wissen, wie viele Tabelleneinträge verfügbar sind, da in dem Fall, daß die Tabelle überläuft, die erforderlichen Übersetzungen gefangen werden und von dem Software-Übersetzer durchgeführt werden.

[0080] Fig. 9 stellt ein zweites Beispiel einer programmierbaren Übersetzungstabelle **104** dar. In diesem Beispiel wird die Übersetzungstabelle in Form eines wahlfrei zugreifbaren Speichers bereitgestellt, wobei der zu übersetzende Bytecode die Eingabe für einen Decoder **106** ist, der den Bytecode als eine Adresse in ein RAM-Array **108** von 4-Bit-Wörtern behandelt, von denen jedes einen eine Operation spezifizierenden Code darstellt. In diesem Fall wird ein eine Operation spezifizierender Code für den Bytecode immer gefunden. Als ein Ergebnis davon verwendet diese Art von Tabelle einen zusätzlichen, eine Operation spezifizierenden Code, der "Codefragment für diesen Bytecode aufrufen" angibt.

[0081] Fig. 10 ist ein schematisches Flußdiagramm, das die Initialisierung und Konfiguration eines Hardware-Übersetzers zur programmierbaren Abbildung mit der Form des Beispiels von Fig. 8 darstellt. In der Praxis werden unterschiedliche Teile von in diesem Flußdiagramm dargestellten Aktionen von Software-Initialisierungsanweisungen bzw. der auf diese Anweisungen reagierenden Hardware durchgeführt.

[0082] Bei Schritt **110** wird eine Anweisung zur Tabelleninitialisierung ausgeführt, die dazu dient, alle vorhandenen Tabelleneinträge zu löschen und einen Zeiger auf den obersten Eintrag in der Tabelle zu setzen. Im Anschluß daran kann Initialisierungscode ausgeführt werden, um Abbildungen in die Übersetzungstabelle mittels Programmanweisungen wie etwa Coprozessor-Register-Ladebefehle zu laden. Die unterschiedlichen Formen dieser Anweisungen zum Laden der Tabelle können abhängig von den speziellen Umständen und der Umgebung variieren. Das Hardware-Übersetzersystem zur programmierbaren Abbildung reagiert auf diese Anweisungen, indem es einen Programmanweisungswert wie einen Java-Bytecode und den Operationswert, der diesen zugeordnet ist, bei Schritt **112** empfängt. Bei Schritt **114** prüft Hardware zum Abfangen nicht unterstützter Operationen, daß der gerade programmierte Operationswert einer ist, der von dem Hardware-Übersetzer zur programmierbaren Abbildung unterstützt wird. Verschiedene Hardware-Übersetzer zur programmierbaren Abbildung können unterschiedliche Sätze von Operationswerten unterstützen und daher mit ihrer eigenen spezifischen Abfang- bzw. Trap-Hardware ausgestattet sein. Die Abfang-Hardware kann relativ einfach sein, wenn ein spezielles System zum Beispiel weiß, daß es die Operationswerte 0,1,2,3,4,5,6,7,8,10, aber nicht **9** unterstützt. Ein Hardware-Vergleicher in Schritt **114** kann den Operationswert auf Gleichheit mit einem Wert von **9** vergleichen und die Programmierung durch Umlenken der Verarbeitung zu Schritt **116** zurückweisen, wenn eine **9** erkannt wird.

[0083] Angenommen daß Schritt **114** anzeigt, daß der Operationswert unterstützt wird, dann prüft Schritt **118**, um festzustellen, ob das Ende der programmierbaren Abbildungstabelle bereits erreicht ist oder nicht. Wenn die programmierbare Abbildungstabelle bereits voll ist, dann geht die Verarbeitung wieder zu Schritt **116** weiter, ohne eine daß eine neue Abbildung hinzugefügt wird. Das Vorsehen von Schritt **118** innerhalb der Hardware bedeutet, daß der Unterstützungscode versuchen kann, die programmierbare Abbildungstabelle ohne Kenntnis davon, wie viele Einträge verfügbar sind, zu programmieren, wobei die Hardware einfach überflüssige Einträge zurückweist. Daher sollte der Programmierer die kritischsten Abbildungen an den Anfang der Tabellenprogrammierung plazieren, um sicherzustellen, daß

diese Plätze einnehmen, die verfügbar sind. Das Vermeiden der Notwendigkeit für den Unterstützungscode zu wissen, wie viele programmierbare Plätze verfügbar sind, bedeutet, daß ein einziger Satz von Unterstützungscode auf mehreren Plattformen betrieben werden kann.

[0084] Angenommen die Tabelle hat einen leeren Eintrag, dann wird die neue Abbildung bei Schritt **120** in diesen Eintrag geschrieben, und der Tabellenzeiger wird dann bei Schritt **122** vorgerückt.

[0085] Bei Schritt **116** prüft das System, ob weitere Programmanweisungscodes in die programmierbare Abbildungstabelle programmiert werden sollen. Schritt **116** ist typischerweise ein Software-Schritt, wobei der Unterstützungscode versucht, so viele Abbildungen, wie er möchte, während der Initialisierung des Systems zu programmieren.

[0086] In dem Fall der Initialisierung einer RAM-Tabelle wie in Fig. 9 abgebildet, kann der oben in Bezug zu Fig. 10 beschriebene Prozeß den folgenden Änderungen unterzogen werden:
daß in Schritt **110** die Tabelle gelöscht wird durch Setzen aller Tabelleneinträge in Array **108** von Fig. 9 auf "Bytecodefragment für diesen Bytecode aufrufen" anstatt durch Setzen des Array **112** in Fig. 8, so daß jeder Eintrag nicht zu irgendeinem Bytecode paßt;
daß in Schritt **110** kein Übersetzungstabellenzeiger zu initialisieren ist;
daß es Schritt **118** nicht gibt, weil es keinen Übersetzungstabellenzeiger gibt;
daß Schritt **120** zu "Operationswert in den durch den Programmanweisungswert angegebenen Tabelleneintrag schreiben" wird; und
daß es Schritt **122** nicht gibt, da es keinen Übersetzungstabellenzeiger gibt.

[0087] Fig. 11 stellt einen Teil einer Verarbeitungspipeline dar, die zur Java-Bytecode-Übersetzung verwendet werden kann. Die Verarbeitungspipeline **124** beinhaltet ein Übersetzungsstadium **126** und ein Java-Decodierstadium **128**. Ein nachfolgendes Stadium **130** könnte eine Vielzahl von verschiedenen Formen annehmen, abhängig von der speziellen Implementierung.

[0088] Wörter von dem Java-Bytecode-Strom werden alternierend in die beiden Hälften des Swing-Puffers **132** geladen. Normalerweise wählt der Multiplexer **133** den aktuellen Bytecode und seine Operanden aus dem Swing-Puffer **132** und liefert ihn über den Multiplexer **137** an das Zwischenspeicherregister bzw. Latch **134**. Wenn der Swing-Puffer **132** leer ist, weil die Pipeline gespült wurde, oder aus irgendeinem anderen Grund, dann wählt der Multiplexer **135** den richtigen Bytecode direkt aus dem ankommenden Wort des Java-Bytecode-Stroms und liefert es an das Latch **134**.

[0089] Der erste Zyklus des Dekodierens für einen Bytecode wird von dem Erstzyklus-Decoder **146** vorgenommen, der auf dem Bytecode in dem Latch **134** agiert. Um Fälle zuzulassen, in denen ein Hardware-unterstützter Bytecode Operanden hat, wählen weitere Multiplexer die Operanden aus dem Swing-Puffer **132** und liefern sie an den Erstzyklus-Decoder **146**. Diese Multiplexer sind in der Figur nicht abgebildet und sind ähnlich dem Multiplexer **133**.

[0090] Typischerweise hat der Erstzyklus-Decoder **146** schwächere Zeitsteuerungsanforderungen für die Operandeneingaben als für die Bytecode-Eingabe, so daß ein Bypass-Pfad ähnlich dem von den Multiplexern **135** und **137** und dem Latch **134** bereitgestellten nicht für die Operanden erforderlich ist.

[0091] Wenn der Swing-Puffer **132** ungenügend viele Operandenbytes für den Bytecode in Latch **134** enthält, dann hält der Erstzyklus-Decoder **146** an, bis genügend Operandenbytes verfügbar sind.

[0092] Die Ausgabe des Erstzyklus-Decoders **146** ist eine ARM-Anweisung (oder ein Satz von den Prozessorkern steuernden Signalen, die eine ARM-Anweisung darstellen), die an das nachfolgende Pipeline-Stadium **130** über den Multiplexer **142** übergeben wird. Eine zweite Ausgabe ist ein eine Operation spezifizierender Code, der in das Latch **138** über den Multiplexer **139** geschrieben wird. Der eine Operation spezifizierende Code beinhaltet ein Bit **140**, das angibt, ob dies ein Einzelzyklus-Bytecode ist.

[0093] Bei dem nächsten Zyklus wird der folgende Bytecode von dem Erstzyklus-Decoder **146** wie zuvor beschrieben dekodiert. Wenn Bit **140** einen Einzelzyklus-Bytecode anzeigt, dann wird der Bytecode dekodiert und steuert das anschließende Pipeline-Stadium **130** wie zuvor beschrieben.

[0094] Wenn Bit **140** stattdessen einen Mehrzyklen-Bytecode anzeigt, dann wird der Erstzyklus-Decoder **146** angehalten und der Mehrzyklen- oder übersetzte Decoder **144** dekodiert den Operation-spezifizierenden Code in Latch **138**, um eine ARM-Anweisung (oder einen Satz von den Prozessorkern steuernden Signalen, die eine ARM-Anweisung darstellen), die der Multiplexer **142** an das anschließende Pipeline-Stadium **130** anstelle der entsprechenden Ausgabe des Erstzyklus-Decoders **146** übergibt. Der Mehrzyklen- oder übersetzte Decoder **144** erzeugt ebenfalls einen weiteren Operation-spezifizierenden Code, der in das Latch **138** über den Multiplexer **139** geschrieben wird, wieder anstelle der entsprechenden Ausgabe des Erstzyklus-Decoders **146**. Dieser weitere Operation-spezifizierende Code enthält auch ein Bit **140**, das angibt, ob dies die letzte ARM-Anweisung ist, die für den Mehrzyklen-Bytecode zu erzeugen ist. Der Mehrzyklen- oder über-

setzte Decoder **144** fährt fort, weitere ARM-Anweisungen wie oben beschrieben zu erzeugen, bis Bit **140** anzeigt, daß die letzte ARM-Anweisung erzeugt wurde, und daraufhin ist der Erstzyklus-Decoder **146** nicht mehr angehalten und erzeugt die erste ARM-Anweisung für den nachfolgenden Bytecode.

[0095] Der oben beschriebene Prozeß wird in dreierlei Art und Weise geändert, wenn der Bytecode in dem Latch **134** übersetzt werden muß. Erstens wird der Bytecode aus dem Swing-Puffer **132** von dem Multiplexer **133** extrahiert und von dem Bytecode-Übersetzer **136** übersetzt, wodurch ein eine Operation spezifizierender Code erzeugt wird, der über den Multiplexer **139** in das Latch **138** geschrieben wird. Dieser eine Operation spezifizierende Code hat Bit **140** gesetzt, um anzuzeigen, daß die letzte ARM-Anweisung nicht für den aktuellen Bytecode erzeugt wurde, so daß der Multiplexer **142** und der Multiplexer **139** die Ausgaben des Mehrzyklen- oder übersetzten Decoder **144** anstatt von denen des Erstzyklus-Decoder **146** beim ersten Zyklus des übersetzten Bytecodes auswählen.

[0096] Zweitens erzeugt der Mehrzyklen- oder übersetzte Decoder **144** alle ARM-Anweisungen, die an das nachfolgenden Pipeline-Stadium **130** zu übergeben sind und ihre entsprechenden weiteren Operation-spezifizierende Codes, die in das Latch **138** zurückzuschreiben sind anstatt nur jene nach dem ersten Zyklus zu erzeugen, wie es für einen Bytecode wäre, der keiner Übersetzung bedarf.

[0097] Drittens, wenn der Bytecode über den Multiplexer **135** direkt in das Latch **134** geschrieben wurde und daher nicht im Swing-Puffer **132** vorhanden war und im vorigen Zyklus nicht von dem Bytecode-Übersetzer **136** übersetzt werden konnte, dann signalisiert der Erstzyklus-Decoder **146** dem Bytecode-Übersetzer **136**, daß er neu starten muß und für einen Zyklus anhält. Dies stellt sicher, daß das Latch **138** einen gültigen, eine Operation spezifizierenden Code für den übersetzten Bytecode hält, wenn der Erstzyklus-Decoder **146** aufhört anzuhalten.

[0098] Man erkennt aus Fig. 11, daß das Bereitstellen eines Übersetzungs-Pipeline-Stadiums ermöglicht, daß die von dem programmierbaren Übersetzungsschritt benötigte Verarbeitung effektiv verborgen oder in die Pipeline gefaltet wird, da die gepufferten Anweisungen nach Bedarf im Vorhinein übersetzt werden und in den Rest der Pipeline einfließen können.

[0099] Man erkennt aus Fig. 11, daß in dieser beispielhaften Ausführungsform der Hardware-Übersetzer der festen Abbildung als prinzipiell von dem Erstzyklus-Decoder **146** und dem Mehrzyklen- oder übersetzten Decoder **144** gebildet betrachtet werden kann, der in dem Modus betrieben wird, in dem er Mehrzyklen-Bytecodes, die der Erstzyklus-Dekodierung durch den Erstzyklus-Decoder **146** unterzogen werden, dekodiert. Der Hardware-Übersetzer der programmierbaren Abbildung kann in diesem Beispiel als von Bytecode-Übersetzer **136** und dem Mehrzyklen- oder übersetzten Decoder **144** gebildet betrachtet werden, der in diesem Fall im Anschluß an die Übersetzung eines programmierbaren Bytecodes betrieben wird. Der Hardware-Übersetzer der festen Abbildung und der Hardware-Übersetzer der programmierbaren Abbildung können in einer breiten Vielfalt von unterschiedlichen Weisen bereitgestellt werden und können sich wesentliche gemeinsame Hardware teilen, während sie ihre unterschiedlichen Funktionen von einem abstrakten Blickwinkel betrachtet behalten. Alle diese unterschiedlichen Möglichkeiten sind von den vorliegend beschriebenen Techniken umfasst.

[0100] Fig. 12 stellt zwei 32-Bit-Befehlswörter **200**, **202** dar, die eine virtuelle Speicherseitengrenze **204** überspannen. Dies kann eine 1 kB-Seitengrenze sein, wenngleich andere Seitengrößen möglich sind.

[0101] Das erste Befehlswort **200** liegt innerhalb einer virtuellen Speicherseite, die sauber innerhalb des virtuellen Speichersystems abgebildet wird. Das zweite Befehlswort **202** liegt innerhalb einer virtuellen Speicherseite, die in diesem Stadium nicht innerhalb des virtuellen Speichersystems liegt. Dementsprechend erhältt eine Zweibyte-Anweisung **206** variabler Länge, die ihr erstes Byte in dem Befehlswort **200** und ihr zweites Byte in dem Befehlswort **202** hat, bei dem einen Abbruch beim Vorab-Holen, der dem zweiten Byte zugeordnet ist. Konventionelle Mechanismen zum Behandeln von Abbrüchen beim Vorab-Holen, die zum Beispiel nur Befehlswort-ausgerichtete Anweisungen unterstützen, sind mäglicherweise nicht in der Lage, mit dieser Situation umzugehen, und könnten zum Beispiel versuchen, das Holen des Befehlswortes **200**, welches das erste Byte der variabel langen Anweisung **206** enthält, zu untersuchen und zu reparieren, statt sich auf das Befehlswort **202** zu konzentrieren, welches das zweite Byte des variabel langen Anweisungswortes **206** enthält, das tatsächlich zu dem Abbruch führte.

[0102] Fig. 13 stellt einen Teil einer Befehlspipeline **208** innerhalb eines Datenverarbeitungssystems zur Verarbeitung von Java-Bytecodes dar, das einen Mechanismus zum Umgang mit Abbrüchen beim Vorab-Holen der in Fig. 12 dargestellten Art beinhaltet. Ein Befehlspuffer enthält zwei Befehlswortregister **210** und **212**, von denen jedes ein 32-Bit Befehlswort speichert. Die Java-Bytecodes sind jeweils 8-Bit lang, begleitet von Null oder mehr Operandenwerten. Eine Gruppe von Multiplexern **214** dient dazu, die passenden Bytes von innerhalb der Befehlswortregister **210** und **212** auszuwählen, abhängig von der aktuellen Java-Bytecode-Zeigerposition, die die

Adresse des ersten Byte der aktuellen zu dekodierenden Java-Bytecode-Anweisung anzeigt.

[0103] Jedem der Befehlswortregister **210** und **212** ist ein entsprechendes Befehlsadreßregister **216**, **218** und Vorab-Holen-Abbruch-Flag-Register **220** und **222** zugeordnet. Diese zugeordneten Register speichern entsprechend die Adresse des Befehlswortes, auf das sie sich beziehen, und, ob ein Abbruch beim Vorab-Holen auftrat oder nicht, als dieses Befehlswort aus dem Speichersystem geholt wurde. Diese Information wird entlang der Pipeline zusammen mit dem Befehlswort selbst übergeben, da diese Information typischerweise weiter hinten in der Pipeline benötigt wird.

[0104] Die Multiplexer **224**, **226** und **228** ermöglichen, daß die Eingangspufferanordnung umgangen werden kann, falls gewünscht. Diese Art von Operation wird oben diskutiert. Es versteht sich, daß die Befehlspipeline **208** aus Gründen der Klarheit nicht alle Eigenschaften der zuvor diskutierten Befehlspipeline zeigt. In ähnlicher Weise zeigt die zuvor diskutierte Befehlspipeline nicht alle Eigenschaften der Befehlspipeline **208**. In der Praxis kann ein System mit einer Kombination der in den beiden dargestellten Befehlspipelines gezeigten Eigenschaften bereitgestellt werden.

[0105] In einem Bytecode-Dekodier-Stadium der Befehlspipeline **208** reagiert ein Bytecode-Decoder **230** auf mindestens einen Java-Bytecode von dem Multiplexer **224** und optional ein oder zwei Operandenbytes von den Multiplexern **226** und **228**, um eine abgebildete Anweisung(en) oder entsprechende Steuersignale zur Übergabe an weitere Stadien in der Pipeline zu erzeugen, um Verarbeitung entsprechend dem dekodierten Java-Bytecode auszuführen.

[0106] Wenn ein Abbruch beim Vorab-Holen der in Fig. 12 dargestellten Art auftrat, dann sind, während der Java-Bytecode selbst gültig sein kann, die nachfolgenden Operandenwerte nicht gültig, und es erfolgt keine korrekte Operation, bis der Abbruch beim Vorab-Holen repariert wird. Ein Bytecode-Ausnahmebedingungs-Generator **232** reagiert sowohl auf Befehlswortadressen aus den Registern **216** und **218** als auch auf Vorab-Holen-Abbruch-Flags aus den Registern **220** und **222**, um das Auftreten der in Fig. 12 dargestellten Art von Situation zu erkennen. Wenn der Bytecode-Ausnahmebedingungs-Generator **232** eine solche Situation erkennt, dann treibt er einen Multiplexer **234** dazu, eine Anweisung oder Steuersignale an die nachfolgenden Stadien, wie sie von dem Bytecode-Ausnahmebedingungs-Generator **232** selbst erzeugt werden, anstelle der von dem Bytecode-Decoder **230** erzeugten auszugeben. Der Bytecode-Ausnahmebedingungs-Generator **232** reagiert auf das Erkennen der Abbruchsituation beim Vorab-Holen von Fig. 12, indem er die Ausführung eines ARM-32.Bit-Codefragmentes anstößt, das den Java-Bytecode, der abgebrochen wird, emuliert, statt der Hardware zu erlauben, diesen Java-Bytecode zu übersetzen. Somit wird die Java-Anweisung variabler Länge **206**, die Gegenstand des Abbruchs beim Vorab-Holen war, nicht selbst ausgeführt, sondern wird von einer Sequenz von 32-Bit-ARM-Anweisungen ersetzt. Die zum Emulieren der Anweisung verwendeten ARM-Anweisungen sind wahrscheinlich Datenabbrüchen ausgesetzt, wenn sie ein oder mehr Operandenbytes laden, wobei diese Datenabbrüche aus demselben Grund auftreten, aus denen die Abbrüche beim Vorab-Holen auftraten, als diese Bytes ursprünglich als Teil des zweiten Befehlswortes **202** geholt wurden, und es ist auch möglich, daß weitere Abbrüche beim Vorab-Holen und Datenabbrüche während der Ausführung des ARM-32-Bit-Codefragments auftreten. Alle diese Abbrüche treten während der Ausführung von ARM-Anweisungen auf und werden somit korrekt von vorhandenen Behandlungsroutinen für Abbruch-Ausnahmebedingungen behandelt.

[0107] Auf diese Weise wird der Abbruch beim Vorab-Holen, der beim Holen der Bytecodes auftrat, unterdrückt (d. h. nicht durch den ARM-Kern durchgereicht). Stattdessen wird eine ARM-Anweisungssequenz ausgeführt, und irgendwelche Abbrüche, die bei diesen ARM-Anweisungen auftreten, werden mittels der vorhandenen Mechanismen behandelt, wodurch schrittweise über den Bytecode, der ein Problem hatte, hinausgegangen wird. Nach Ausführung der emulierenden ARM-Anweisungen, die zum Ersetzen des Bytecode mit einem Abbruch verwendet werden, kann die Ausführung der Bytecodes wieder aufgenommen werden.

[0108] Wenn der Bytecode selbst einen Abbruch beim Vorab-Holen erleidet, dann wird eine mit einem Abbruch beim Vorab-Holen markierte ARM-Anweisung an den Rest der ARM-Pipeline übergeben. Falls und wenn sie das Ausführen-Stadium der Pipeline erreicht, verursacht sie, daß eine Ausnahmebedingung "Abbruch beim Vorab-Holen" auftritt: dies ist eine vollständig standardgemäße Methode zur Behandlung von Abbrüchen beim Vorab-Holen bei ARM-Anweisungen.

[0109] Wenn der Bytecode keinen Abbruch beim Vorab-Holen erleidet, aber ein oder mehrere seiner Operanden einen Abbruch erfahren, wie in Fig. 12 gezeigt, dann wird das Software-Codefragment für diesen Bytecode aufgerufen. Irgendwelche ARM-Anweisungen, die an den Rest der ARM-Pipeline übergeben werden, um das Aufrufen des Codefragmentes zu veranlassen, werden nicht mit einem Abbruch beim Vorab-Holen markiert, und werden somit normal ausgeführt, falls und wenn sie das Ausführen-Stadium der Pipeline erreichen.

[0110] Fig. 14 stellt einen logischen Ausdruck der

Art dar, die von dem Bytecode-Ausnahmebedingungs-Generator **232** verwendet werden kann, um die in Fig. 12 dargestellte Art von Situation zu erkennen. "Half1" bezeichne, welche Hälfte auch immer des Swing-Puffers in Fig. 13 (die Blöcke **210**, **216**, **220** bilden eine Hälfte, während die Blöcke **212**, **218**, **222** die andere Hälfte bilden, wie durch die gestrichelten Linien um diese Elemente in Fig. 13 angegeben) gerade das erste Befehlswort hält (**200** in Fig. 12) und "Half2" die andere Hälfte des Swing-Puffers, die das zweite Befehlswort (**202** in Fig. 12) halten. PA(Half1) bedeute den Inhalt desjenigen der Blöcke **220** und **222**, welcher auch immer in Half1 ist, und in ähnlicher Weise für Half2.

[0111] Dann sind die Anzeichen für die in Fig. 12 dargestellte Situation, daß PA(Half1) falsch ist, PA(Half2) wahr ist und der Bytecode plus seine Operanden die Grenze zwischen den beiden Swing-Pufferhälften überspannen. (Die Tatsache, daß es dort eine markierte Seitengrenze gibt, hat nur den Grund, daß es normalerweise eine Voraussetzung dafür ist, daß die beiden PA()-Werte voneinander abweichen können.)

[0112] In bevorzugten Ausführungen wie denjenigen, in denen die Swing-Pufferhälften jeweils ein Wort speichern und Hardware-unterstützte Bytecodes auf maximal 2 Operanden beschränkt sind, ist die Formel zum Bestimmen, ob der Bytecode und seine Operanden die Grenze überspannen:
((Anzahl von Operanden = 1) UND (bcaddr[1:0] = 11))
ODER ((Anzahl von Operanden = 2) UND (bcaddr[1] = 1))
wobei bcaddr die Adresse des Bytecodes ist. Die erlaubt es, den in Fig. 14 abgebildeten logische Ausdruck abzuleiten.

[0113] Andere Techniken zum Identifizieren eines Abbruchs beim Vorab-Holen können verwendet werden wie etwa, daß eine Anweisung variabler Länge innerhalb eines vorher festgelegten Abstandes von einer Speicherseitengrenze beginnt.

[0114] Fig. 15 stellt die Struktur des der Java-Bytecode-Übersetzung zugeordneten Unterstützungscodes schematisch dar. Dies ist ähnlich der zuvor diskutierten Figur, aber in diesem Fall stellt sie den Einschluß bzw. die Hinzunahme der Zeiger auf die Codefragmente zur Behandlung einer Bytecode-Ausnahmebedingung dar, die von den Bytecode-Ausnahmebedingungs-Ereignissen angestoßen bzw. ausgelöst werden. Somit hat jeder Java-Bytecode ein zugeordnetes ARM-Codefragment, das seine Operation emuliert. Darüber hinaus hat jede Bytecode-Ausnahmebedingung, die vorkommen kann, einen zugeordneten Teil eines ARM-Ausnahmebedingungs-Behandlungscodes. In dem dargestellten Fall wird eine Behandlungsroutine **236** für einen Abbruch beim Vorab-Holen von Bytecodes bereitgestellt, um beim Erkennen der oben beschriebenen Art von Abbruch beim Vorab-Holen von dem Bytecode-Ausnahmebedingungs-Generator **232** ausgelöst zu werden. Dieser Abbruchbehandlungscode **236** funktioniert so, daß er den Bytecode am Anfang der Anweisung variabler Länge identifiziert, der Anlaß dafür war, daß er ausgelöst wurde, und dann das zugehörige Emulation-Codefragment für diesen Bytecode innerhalb der Sammlung von Codefragmenten aufruft.

[0115] Fig. 16 ist ein Flußdiagramm, das die Funktion des Bytecode-Ausnahmebedingungs-Generators **232** und der anschließenden Verarbeitung schematisch darstellt. Schritt **238** dient der Feststellung, ob der Ausdruck von Fig. 14 wahr ist oder nicht. Wenn der Ausdruck falsch ist, dann endet dieser Vorgang.

[0116] Wenn Schritt **238** die in Fig. 12 dargestellte Art von Situation angezeigt hat, dann wird Schritt **246** ausgeführt, der eine Abbruchausnahmebedingung beim Vorab-Holen eines Bytecodes auslöst, die von dem Bytecode-Ausnahmebedingungs-Generator **232** eingeleitet wurde. Der Bytecode-Ausnahmebedingungs-Generator **232** kann einfach die Ausführung der ARM- Code-Behandlung **236** eines Abbruchs beim Vorab-Holen eines Bytecodes auslösen. Die Abbruchbehandlung **236** dient bei Schritt **248** dazu, den Bytecode zu identifizieren, der die Anweisung variabler Länge einleitet, und löst dann bei Schritt **250** die Ausführung des Codefragmentes von ARM-Anweisungen aus, das diesen identifizierten Bytecode emuliert.

[0117] Der oben beschriebene Mechanismus zum Umgang mit Abbrüchen beim Vorab-Holen funktioniert gut in Situationen, in denen es vier oder weniger Operanden gibt (d. h. fünf oder weniger Bytes insgesamt), ansonsten kann es möglich sein, daß ein Bytecode und seine Operanden über den zweiten Puffer hinausreichen. In der Praxis haben alle Bytecodes, für die es vorgezogen wird, einen Hardwarebeschleunigungsmechanismus bereitzustellen 0, 1 oder 2 Operanden, wobei der Rest der Bytecodes in allen Fällen in Software abgehandelt wird, hauptsächlich aufgrund ihrer Komplexität.

[0118] Fig. 17 stellt ein Betriebssystem 300 zum Steuern einer Mehrzahl von Benutzermodus-Prozessen **302**, **304**, **306** und **308** dar. Das Betriebssystem **300** arbeitet in einem Supervisor-Modus und die anderen Prozesse **302**, **304**, **306** und **308** arbeiten in einem Benutzer-Modus, wobei sie geringere Zugriffsrechte auf Konfigurationsteuerparameter des Systems als das Betriebssystem **300** haben, das im Supervisor-Modus arbeitet.

[0119] Wie in Fig. 17 dargestellt beziehen sich die Prozesse **302** und **308** auf verschiedene virtuelle Java-Maschinen. Für jede dieser virtuellen Java-Machi-

nen **302**, **308** werden ihre eigene Konfigurationsdaten gebildet aus Bytecode-Übersetzungs-Abbildungsdaten **310**, **312** und Konfigurationsregisterdaten **314**, **316**. In der Praxis wird zugestandenermaßen ein einziger Satz von Java-Beschleunigungshardware zur Ausführung beider Prozesse **302**, **308** bereitgestellt, aber wenn diese unterschiedlichen Prozesse die Java-Beschleunigungshardware verwenden, benötigt jeder von ihnen, daß sie mit ihren zugeordneten Konfigurationsdaten **310**, **312**, **314**, **316** konfiguriert ist. Daher sollte in dem Fall, daß das Betriebssystem **300** die Ausführung zu einem Prozeß umschaltet, der die Java-Beschleunigungshardware verwendet, der von dem vorigen Prozeß, der diese Hardware verwendet hat, verschieden ist, die Java-Beschleunigungshardware dann neu initialisiert und neu konfiguriert werden. Das Betriebssystem **300** vollzieht diese erneute Initialisierung und erneute Konfiguration der Java-Beschleunigungshardware selbst nicht, sondern zeigt an, daß es erfolgen sollte, indem es einen Kofiguration-Ungültig-Indikator, der der Java-Beschleunigungshardware zugeordnet ist, auf einen ungültigen Zustand setzt.

[0120] Fig. 18 stellt ein Datenverarbeitungssystem **318** schematisch dar, das einen Prozessorkern **320** mit einem maschineneigenen Befehlssatz (z. B. den ARM-Befehlssatz) und zugeordnete Java-Beschleunigungshardware **322** beinhaltet. Ein Speicher **324** speichert Computerprogrammcode, der in der Form von ARM-Anweisungen oder Java-Bytecodes vorliegen kann. In dem Fall von Java-Bytecodes werden diese durch die Java-Beschleunigungshardware **322** hindurch gereicht, die dazu dient, sie in einen Strom von ARM-Anweisungen zu übersetzen (oder von Steuersignalen, die ARM-Anweisungen entsprechen), der dann von dem Prozessorkern **320** ausgeführt werden kann. Die Java-Beschleunigungshardware **322** beinhaltet eine Bytecode-Übersetzungstabelle **326**, die für jede virtuelle Java-Maschine programmiert werden muß, für die sie Java-Bytecodes ausführen soll. Ferner sind ein Konfigurationsdatenregister **328** und ein Betriebssystem-Steuerregister **330** innerhalb der Java-Beschleunigungshardware **322** zum Steuern ihrer Konfiguration vorgesehen. Innerhalb des Betriebssystem-Steuerregisters **330** ist ein Konfiguration-Gültig-Indikator in der Form eines Flag CV enthalten, das anzeigt, wenn es gesetzt ist, daß die Konfiguration der Java-Beschleunigungshardware **322** gültig ist, und wenn es nicht gesetzt ist, daß sie ungültig ist.

[0121] Die Java-Beschleunigungshardware **322** reagiert, wenn sie einen Java-Bytecode auszuführen versucht, auf den Konfiguration-Gültig-Indikator, um eine Konfiguration-Ungültig-Ausnahmebedingung auszulösen, wenn der Konfiguration-Gültig-Indikator der Situation entspricht, daß die Konfigurationsdaten für die Java-Beschleunigungshardware **322** in einer ungültigen Form sind. Die Behandlungsroutine der Konfiguration-Ungültig-Ausnahmebedingung kann eine ARM-Coderoutine sein, die in einer Weise ähnlich der oben für die Behandlungsroutine eines Abbruchs beim Vorab-Holen bereitgestellt werden kann. Ein Hardware-Mechanismus ist innerhalb der Java-Beschleunigungshardware **322** vorgesehen, der den Konfiguration-Gültig-Indikator in eine Form bringt, die anzeigtn, daß die Konfigurationsdaten gültig sind, wenn die Konfigurations-Ausnahmebedingung ausgelöst wird und bevor die neuen gültigen Konfigurationsdaten tatsächlich an die Stelle geschrieben wurden. Während es der Intuition zu widersprechen scheint, den Konfiguration-Gültig-Indikator in dieser Weise zu setzen, bevor die Konfigurationsdaten tatsächlich geschrieben wurden, hat dieser Ansatz bedeutende Vorteile, indem er in der Lage ist, Probleme zu vermeiden, die bei einer Prozessumschaltung während des Setzens der Konfigurationsdaten entstehen können. Die Behandlungsroutine für die Konfigurations-Ausnahmebedingung richtet dann die benötigten Konfigurationsdaten für die Java Virtual Machine ein, zu der sie gehört, indem sie wie zuvor diskutiert Einträge in die Bytecode-Übersetzungstabelle und jegliche anderen Konfigurationsdatenregisterwerte **328** wie benötigt schreibt. Der Konfigurationsausnahmebedingungscode muß sicherstellen, daß das Schreiben der Konfigurationsdaten abgeschlossen ist, bevor irgendeine andere Aufgabe von der Java-Beschleunigungshardware **322** in Angriff genommen wird.

[0122] Fig. 19 stellt den Betrieb des Betriebssystems **300** schematisch dar. Bei Schritt **332** wartet das Betriebssystem, um eine Prozeßumschaltung zu erkennen. Wenn eine Prozeßumschaltung erkannt wird, stellt Schritt **334** fest, ob der neue Prozeß einer ist, der die Java-Beschleunigungshardware **322** (auch Jazelle genannt, wie zuvor erwähnt) verwendet oder nicht. Wenn die Java-Beschleunigungshardware **322** nicht verwendet wird, dann schreitet die Verarbeitung zu Schritt **336** fort, bei dem die Java-Beschleunigungshardware **322** gesperrt wird, bevor zu Schritt **339** übergegangen wird, bei dem die Ausführung auf den neuen Prozeß übertragen wird. Wenn die Java-Beschleunigungshardware **322** verwendet wird, dann schreitet die Verarbeitung zu Schritt **338** fort, bei dem eine Feststellung getroffen wird, ob der neue Prozeß, der aufgerufen wird, derselbe ist wie der gespeicherte aktuelle Besitzer der Java-Beschleunigungshardware **322**, wie von dem Betriebssystem **300** aufgezeichnet, oder nicht. Wenn sich der Besitzer nicht geändert hat (d. h. der neue Prozeß ist tatsächlich derselbe wie der letzte Prozeß, der die Java-Beschleunigungshardware **322** benutzt hat), dann geht die Verarbeitung zu Schritt **337** weiter, bei dem die Java-Beschleunigungshardware **322** vor dem Weitergehen zu Schritt **339** freigegeben bzw. entsperrt wird. Wenn der neue Prozeß nicht der gespeicherte aktuelle Besitzer ist, dann schreitet die Verarbeitung zu Schritt **340** fort, bei dem der Konfigurati-

on-Gültig-Indikator gesetzt wird, um anzuzeigen, daß die aktuelle Konfiguration der Java-Beschleunigungshardware **322** nicht gültig ist. Dies ist die Grenze der Verantwortlichkeit des Betriebssystems **300** zum Verwalten dieser Konfigurationsänderung, wobei das tatsächliche Aktualisieren der Konfigurationsdaten als eine Aufgabe der Java-Beschleunigungshardware **322** selbst überlassen ist, die mit ihren eigenen Behandlungsmechanismen für Ausnahmebedingungen arbeitet.

[0123] Nach Schritt **340** dient Schritt **342** dazu, den gespeicherten aktuellen Besitzer als den neuen Prozeß zu aktualisieren, bevor die Ausführungskontrolle an Schritt **337** übergeben wird und dann an Schritt **339**.

[0124] Fig. 20 stellt die Operationen dar, die von der Java-Beschleunigungshardware **322** durchgeführt werden. Bei Schritt **344** wartet die Java-Beschleunigungshardware **322** darauf, einen Bytecode zum Ausführen zu erhalten. Wenn ein Bytecode erhalten wird, prüft die Hardware mittels Schritt **346**, daß der Konfiguration-Gültig-Indikator anzeigt, daß die Konfiguration der Java-Beschleunigungshardware **322** gültig ist. Wenn die Konfiguration gültig ist, dann geht die Verarbeitung zu Schritt **348** weiter, bei dem der erhaltene Bytecode ausgeführt wird.

[0125] Wenn die Konfiguration ungültig ist, dann geht die Verarbeitung zu Schritt **350** weiter, bei dem die Java-Beschleunigungshardware **322** einen Hardwaremechanismus verwendet, um den Konfiguration-Gültig-Indikator zu setzen, um anzuzeigen, daß die Konfiguration gültig ist. Dies könnte auch von einer Programmanweisung innerhalb der Ausnahmebehandlungsroutine getan werden, wenn gewünscht. Schritt **352** dient als Auslöser eine Konfiguration-Ungültig-Ausnahmebedingung. Die Behandlungsroutine der Konfiguration-Ungültig-Ausnahmebedingung kann als eine Kombination einer Tabelle von Zeigern auf Codefragmente und zugeordneten Codefragmenten zum Behandeln jeder in Frage kommenden Ausnahmebedingung wie Software-Emulation einer Anweisung, eines Abbruchs beim Vorab-Holen (beides wurde oben diskutiert) wie in diesem Fall, oder einer Konfigurations-Ausnahmebedingung bereitgestellt werden.

[0126] Schritt **354** dient zum Ausführen des ARM-Codes, der die Konfiguration-Ungültig-Ausnahmebedingung ausmacht, und dieser dient dazu, die benötigten Konfigurationsdaten in die Java-Beschleunigungshardware **322** zu schreiben. Dieser ARM-Code kann die Form einer Sequenz von Coprozessor-Register-Schreibvorgängen annehmen, um sowohl die programmierbare Übersetzungstabelle **326** als auch andere Konfigurationsregister **330** zu besetzen. Nach Schritt **354** springt Schritt **356** zurück in das Java-Bytecode-Programm, um die Ausführung des ursprünglichen Bytecodes erneut zu versuchen.

[0127] Wenn eine Prozeßumschaltung während des Schrittes **354** oder des Schrittes **358** eintritt, ist es möglich, daß das bisherige Einrichten der Konfiguration durch den anderen Prozeß ungültig gemacht wird und der Konfiguration-Gültig-Indikator von dem Betriebssystem gelöscht wird. In der Prozedur von Fig. 20 führt das dazu, daß die 344-346-350-352-354-Schleife erneut durchlaufen wird, d. h. daß die Neukonfiguration vom Beginn erneut versucht wird. Wenn der Bytecode eventuell tatsächlich ausgeführt wird, ist die Konfiguration garantiert gültig.

[0128] Fig. 21 stellt ein Datenverarbeitungssystem wie in Fig. 1 dar, das ferner ein Gleitkomma-Subsystem beinhaltet. Wenn eine nicht abgehandelte Gleitkommaoperation auftritt, sieht das Gleitkomma-Subsystem Mechanismen vor, die nicht abgehandelte Gleitkommaoperation in ARM-Code zu behandeln.

[0129] Ein Beispiel eines solchen Subsystems ist das VFP-Software-Emulator-System von ARM Limited aus Cambridge, England. In dem Fall des VFP-Software-Emulator-Systems werden alle Gleitkommaoperationen als nicht abgehandelte Gleitkommaoperationen behandelt, da keine Hardware verfügbar ist, um die Gleitkommaoperationen durchzuführen. Alle Gleitkommaoperationen werden daher mittels der vorgesehenen Mechanismen zum Emulieren des Verhaltens des VFP in ARM-Code behandelt.

[0130] Im Falle eines solchen Systems sind nicht abgehandelte Gleitkommaoperationen präzise, das heißt der Zeitpunkt des Erkennens einer nicht abgehandelten Gleitkommaoperation ist derselbe wie der Zeitpunkt des Auftretens der nicht abgehandelten Gleitkommaoperation.

[0131] Fig. 22 stellt ein Datenverarbeitungssystem wie in den Fig. 1 und Fig. 21 dar, welches darüber hinaus ein Gleitkommaoperationsregister und ein Zustandsflag "nicht abgehandelte Operation" beinhaltet.

[0132] Ein Beispiel eines solchen Subsystems ist das VFP-Hardware-Subsystem von ARM Limited aus Cambridge, England. In dem Fall des VFP-Hardware-Subsystems werden nur bestimmte Typen von Gleitkommaoperationen als nicht abgehandelte Gleitkommaoperationen behandelt, der Rest wird von der VFP-Hardware behandelt.

[0133] Die Klasse von Operationen, die Gegenstand nicht abgehandelter Gleitkommaoperationen sein können, umfaßt:

- Division durch Null
- Operationen, die ein NaN nach sich ziehen
- Operationen, die eine Unendlichkeit nach sich

ziehen
• Operationen, die denormalisierte Zahlen nach sich ziehen

[0134] In dem Fall eines solchen Systems kann eine nicht abgehandelte Gleitkomma Operation unpräzise sein, das heißt der Zeitpunkt des Erkennens einer nicht abgehandelten Gleitkommaoperation ist nicht notwendigerweise derselbe wie der Zeitpunkt des Auftretens der nicht abgehandelten Gleitkommaoperation.

[0135] Eine nicht abgehandelte VFP-Operation tritt auf, wenn der VFP-Coprozessor es ablehnt, eine VFP-Anweisung zu akzeptieren, die normalerweise Teil eines ARM-Anweisungsstromes bilden würde, aber in der Gegenwart eines in Fig. 1 gezeigten Bytecode-Übersetzers das Ergebnis eines Bytecodes sein kann, der in eine Kombination von ARM- und VFP-Anweisungen übersetzt wurde.

[0136] In dem Fall, daß eine nicht abgehandelte VFP-Operation als Teil eines ARM-Anweisungsstromes auftritt, soll der ARM-Mechanismus zum Behandeln der nicht abgehandelten VFP-Operation eine Nicht-Definierte-Anweisung-Ausnahmebedingung erzeugen und die Behandlungsroutine für nicht definierte Anweisungen ausführen, die auf dem Nicht-Definierte-Anweisung-Vektor installiert ist.

[0137] Im Fall des VFP-Software-Emulator-Systems werden alle VFP-Operationen als nicht abgehandelte VFP-Operationen behandelt, und derselbe ARM-Mechanismus wird angewandt, eine Nicht-Definierte-Anweisung-Ausnahmebedingung erzeugt und die Behandlungsroutine für nicht definierte Anweisungen ausgeführt.

[0138] Wenn die nicht abgehandelte VFP-Operation als Teil eines ARM-Anweisungsstromes auftritt, kann die Behandlungsroutine für nicht definierte Anweisungen durch Inspizieren des Anweisungsstromes erkennen, daß die Anweisung, welche die nicht abgehandelte VFP-Operation verursacht hat, tatsächlich eine VFP-Operation war und nicht irgendeine andere Art von nicht definierter Anweisung, und beim Ausführen der Behandlungsroutine für nicht definierte Anweisungen in einem privilegierten Modus kann sie die benötigten Coprozessor-Anweisungen ausgeben, um jedweden internen Zustand zu extrahieren, den sie von dem VFP-Coprozessor benötigt, und die erforderliche Anweisung in Software zu Ende führen. Die Behandlungsroutine für nicht definierte Anweisungen wird sowohl die in dem ARM-Anweisungsstrom identifizierte Anweisung als auch den internen Zustand des VFP verwenden, um die nicht abgehandelte Operation zu behandeln.

[0139] Auf vielen VFP-Implementierungen ist die Anweisung, die die nicht abgehandelte Operation verursacht hat, möglicherweise nicht dieselbe wie die Anweisung, die ausgeführt wurde, als die nicht abgehandelte Operation entdeckt wurde. Die nicht abgehandelte Operation kann von einer Anweisung verursacht worden sein, die früher abgesetzt wurde, parallel mit nachfolgenden ARM-Anweisungen ausgeführt wurde, aber auf einen nicht behandelten Zustand antrifft. Der VFP signalisiert dies, indem er es ablehnt, eine nachfolgende VFP-Anweisung entgegen zu nehmen, wodurch er erzwingt, daß in die VFP-Behandlungsroutine für undefinierte Anweisungen hineingegangen wird, die den VFP befragen kann, um den ursprünglichen Grund der nicht abgehandelten Operation herauszufinden.

[0140] Wenn Jazelle in ein System integriert ist, das ein VFP-Subsystem enthält, gilt das Folgende:
• Java-Gleitkomma-Anweisungen werden übersetzt durch Absetzen der entsprechenden VFP-Anweisung direkt innerhalb des Kerns mittels eines Satzes von Signalen mit einer direkten Entsprechung zu VFP-Anweisungen.
• Der VFP kann eine Bedingung "nicht abgehandelte Operation" signalisieren, wenn er auf eine nicht abgehandelte Operation stößt.
• Jazelle fängt das Signal "nicht abgehandelte Operation" ab und verhindert damit, daß es an den Kern gesendet wird und verhindert, daß die Behandlungsroutine für nicht definierte Anweisungen ausgeführt wird, was passieren würde, wenn eine VFP-Anweisung in einem ARM-Anweisungsstrom eine nicht korrekte Operation signalisiert hat. Jazelle erzeugt stattdessen eine Jazelle-VFP-Ausnahmebedingung, die von dem Jazelle-VM-Unterstützungscode behandelt wird.

[0141] Der VM-Unterstützungscode sollte beim Antreffen einer solchen Jazelle-VFP-Ausnahmebedingung eine VFP-'No-Operation'-Anweisung ausführen, d. h. irgendeine VFP-Anweisung, die den Jazelle-Zustand unberührt läßt wie eine FMRX Rd, FPSCR-Anweisung. Dies synchronisiert die VPF-Hardware mit dem Unterstützungscode und führt die Operation irgendeiner VFP-Operation zu Ende, die angezeigt wird von dem Gleitkommaoperationsregister in Verbindung mit dem Zustandsflag "nicht abgehandelte Operation", das in diesem Fall gesetzt werden sollte, wenn gerade eine nicht abgehandelte Operation angetroffen wurde. Sobald die Operation abgeschlossen ist, wird das Zustandsflag "nicht abgehandelte Operation" gelöscht.

[0142] Der Ansatz nutzt die Tatsache aus, daß die von Jazelle abgesetzten Anweisungssequenzen erneut gestartet werden können, wie in der gleichzeitig anhängigen britischen Patentanmeldung Nummer 0024402.0, registriert am 5. Oktober 2000, beschrieben, die hier in ihrer Gesamtheit durch diese Bezugnahme einbezogen ist. Die Verwendung der in dem obigen Dokument beschriebenen Technik in Verbin-

dung mit dieser Technik ermöglicht es, daß die Anweisung, die das Erzeugen der VFP-Anweisung veranlaßte, die zu der nicht abgehandelten Operation führte, erneut gestartet werden kann.

[0143] Fig. 23 stellt für jede der Java-Gleitkommaoperationen die entsprechenden VFP-Anweisungen dar, die von dem Java-Bytecode-Übersetzer abgesetzt werden. Man beachte, daß nur die VFP-Anweisungen, die abgesetzt werden, dargestellt sind, wobei der Java-Bytecode-Übersetzer (eine) zusätzliche(n) ARM-Anweisung(en) in Verbindung mit den VFP-Anweisungen absetzen kann. Der Jazelle-Bytecode-Übersetzer kann auch zusätzliche VFP-Lade- und -Speicheranweisungen absetzen, um Gleitkomma-Werte zu laden oder zu speichern.

[0144] Fig. 24 stellt eine Sequenz von Anweisungen oder Signalen, die Anweisungen entsprechen, dar, die von dem Jazelle-Bytecode-Übersetzer für die Sequenz von Java-Bytecodes, die aus einem 'dmul'-Bytecoe gefolgt von einem 'dcmpg'-Bytecode besteht, abgesetzt werden könnte.

[0145] Die dargestellte Sequenz würde auftreten, wenn eine (dmul, dcmpg)-Bytecode-Sequenz auszuführen wäre zu einem Zeitpunkt, zu dem die doppeltgenauen Register D0, D1 und D2 das dritte von oben, zweite von oben bzw. oberste Element des Java-Ausführungsstack enthielten, und dieses Ganzzahl-Ergebnis der Bytecode-Sequenz in das Ganzzahl-Register R0 plaziert werden soll.

[0146] Die Fig. 25, Fig. 27, Fig. 29 und Fig. 30 stellen die Sequenz von Operationen dar, wenn eine nicht abgehandelte Gleitkommaoperation an verschiedenen Punkten in der übersetzten Anweisungssequenz auftritt. Die Fig. 25 und Fig. 29 stellen die Sequenz von Operationen dar, wenn die nicht abgehandelte Gleitkommaoperation durch die FMULD-Anweisung verursacht ist. Die Fig. 27 und Fig. 30 stellen die Sequenz von Operationen dar, wenn eine nicht abgehandelte Gleitkommaoperation durch die FCMPD-Anweisung verursacht ist. Die Fig. 25 und Fig. 27 stellen die Sequenz von Operationen dar, wenn die Signalisierung von nicht abgehandelten Gleitkommaoperationen unpräzise ist. Die Fig. 29 und Fig. 30 stellen die Sequenz von Operationen dar, wenn die Signalisierung von nicht abgehandelten Gleitkommaoperationen präzise ist.

[0147] Wie man erkennt, gibt es vier mögliche Sequenzen von Ereignissen:

1) Fig. 25: Erfassung von unpräzisen, nicht abgehandelten Operationen, Java-Bytecode, der die nicht abgehandelte Operation signalisiert, ist nicht derselbe, der die nicht abgehandelte Operation verursacht hat.

2) Fig. 27: Erfassung von unpräzisen, nicht abgehandelten Operationen, Java-Bytecode, der die nicht abgehandelte Operation signalisiert ist derselbe, der sie verursacht hat trotz der Tatsache, daß das System eine Erfassung von unpräzisen, nicht abgehandelten Operationen verwendet. Dies liegt daran, daß der zweite Java-Bytecode 'dcmpg'2 VFP-Anweisungen für einen Java-Bytecode absetzt, von denen der erste die nicht abgehandelte Operation verursacht und von denen der zweite sie signalisiert.

3) Fig. 29: Präzise Entdeckung von nicht abgehandelten Operationen, Java-Bytecode, der die nicht abgehandelte Operation signalisiert ist derselbe, der sie verursacht hat.

4) Fig. 30: Präzise Entdeckung von nicht abgehandelten Operationen, Java-Bytecode, der die nicht abgehandelte Operation signalisiert ist derselbe, der sie verursacht hat, jedoch ist nicht bekannt, welche der beiden als Ergebnis der Ausführung des 'dcmpg'-Bytecodes abgesetzten VFP-Anweisungen tatsächlich die nicht abgehandelte Operation verursacht und signalisiert hat.

[0148] Die Kombination der oben erwähnten Neustart-Technik mit dieser Technik ermöglicht es, daß alle diese möglichen Sequenzen von Ereignissen korrekt behandelt werden.

[0149] Die Fig. 26 und Fig. 28 stellen den Zustand des Gleitkommaoperationsregisters und des Flag für einen Zustand "nicht abgehandelte Operation" zu einem Zeitpunkt dar, unmittelbar nachdem die nicht abgehandelte Operation verursacht wurde, entsprechend der in den Fig. 25 bzw. 27 dargestellten Sequenz von Operationen.

[0150] Es sollte Bezug genommen werden auf GB-A-2367 652, GB-A-2367 653, GB-A-2367 654, GB-A-2367 651, GB-A-2367 658 und US-A-2002063462, die ebenfalls ein System zur Übersetzung von Java-Bytecode beschreiben.

Patentansprüche

1. Vorrichtung zur Verarbeitung von Daten unter der Steuerung von Programmanweisungen bzw. -befehlen aus einem ersten Satz von Programmbefehlen oder Programmanweisungen bzw. -befehlen aus einem zweiten Satz von Programmbefehlen, wobei die Vorrichtung aufweist:
einen Softwareübersetzer von Befehlen, der derart betreibbar ist, daß er einen Programmbefehl des zweiten Satzes von Programmbefehlen als eine Sequenz von Programmbefehlen des ersten Satzes von Programmbefehlen übersetzt, welche mit einem die Sequenz abschließenden Befehl endet, und
einem Anweisungs- bzw. Befehlsdecoder, der auf den die Sequenz abschließenden Befehl reagiert:
(i) falls eine Befehlsausführungseinheit in Hardware für den zweiten Satz von Befehlen nicht verfügbar ist, Initiieren der Übersetzung eines nächsten Pro-

grammbefehls des zweiten Satzes von Befehlen unter Verwendung des Softwareübersetzers für Befehle, und
(ii) falls die Befehlsausführungseinheit in Hardware für den zweiten Satz von Befehlen bzw. Anweisungen verfügbar ist, Auslösen der Ausführung des nächsten Programmbefehls des zweiten Satzes von Befehlen unter Verwendung der Befehlsausführungseinheit in Hardware.

2. Vorrichtung nach Anspruch 1, wobei die Befehlsausführungseinheit in Hardware nicht verfügbar ist, wenn:
(i) die Befehlsausführung in Hardware nicht freigeschaltet ist, oder
(ii) eine Befehlsausführungseinheit in Hardware nicht vorhanden ist.

3. Vorrichtung nach einem der Ansprüche 1 oder 2, wobei der Befehlsdecoder auf zumindest ein Flag der Befehlsausführungseinheit in Hardware reagiert, um zu erfassen, ob eine Befehlsausführungseinheit in Hardware verfügbar ist oder nicht.

4. Vorrichtung nach einem der Ansprüche 1, 2 oder 3, wobei der eine Sequenz abschließende Befehl eine Startadresse für die Softwareübersetzung innerhalb des in Software implementierten Befehlsübersetzers einer Sequenz von Programmbefehlen des ersten Satzes von Befehlen angibt, welche dazu dienen, den nächsten Programmbefehl des zweiten Satzes von Befehlen zu übersetzen, wobei der in Software implementierte Befehlsübersetzer die Sequenz verwendet, falls die Befehlsausführungseinheit in Hardware nicht verfügbar ist.

5. Vorrichtung nach Anspruch 4, wobei der die Sequenz abschließende Befehl einen Operanden enthält, welcher ein Register angibt, das die Startadresse für die Softwareübersetzung speichert.

6. Vorrichtung nach Anspruch 4, wobei der die Sequenz abschließende Befehl ein vorbestimmtes Register verwendet, um die Startadresse für die Übersetzung in Software zu speichern.

7. Vorrichtung nach einem der vorstehenden Ansprüche, wobei der die Sequenz abschließende Befehl die Startadresse der Hardwareausführung angibt, welche auf den nächsten Befehl des zweiten Satzes von Befehlen hinweist, wobei die Startadresse für die Ausführung in Hardware durch die Befehlsausführungseinheit in Hardware verwendet wird, um auf den nächsten Befehl des zweiten Satzes von Befehlen zuzugreifen, wenn die Befehlsausführungseinheit in Hardware verfügbar ist.

8. Vorrichtung nach Anspruch 7, wobei der die Sequenz abschließende Befehl ein vorbestimmtes Register verwendet, um die Startadresse der Hardwareausführung zu speichern.

9. Vorrichtung nach Anspruch 7, wobei der die Sequenz abschließende Befehl einen Operanden enthält, welcher ein Register angibt, das die Startadresse der in Hardware erfolgenden Ausführung speichert.

10. Vorrichtung nach einem der vorstehenden Ansprüche, wobei der Befehlsübersetzer in Software eine Mehrzahl von Sequenzen von Programmbefehlen aus dem ersten Satz von Befehlen aufweist, wobei die Sequenzen dieser Mehrzahl jeweils Programmbefehlen des zweiten Satzes von Befehlen entsprechen.

11. Vorrichtung nach Anspruch 10, wobei der in Software implementierte Befehlsübersetzer eine Tabelle von Zeigern auf die Mehrzahl von Sequenzen enthält.

12. Vorrichtung nach Anspruch 11, wobei ein Eintrag innerhalb der Tabelle durch einen Programmbefehl des zweiten Satzes von Befehlen erzeugt wurde, die zu übersetzen sind.

13. Vorrichtung nach einem der Ansprüche 11 oder 12, wobei eine Basisadresse der Tabelle von Zeigern innerhalb eines Basisadreßregisters gespeichert ist.

14. Vorrichtung nach einem der vorstehenden Ansprüche, welche einen Prozessorkern aufweist, wobei die Programmbefehle des ersten Satzes von Befehlen natürliche bzw. inhärente Programmbefehle sind, die durch den Prozessorkern ausgeführt werden.

15. Vorrichtung nach Anspruch 14, wobei die Befehlsausführungseinheit in Hardware Befehle in Java-Bytecode zumindest als eine Darstellung eines oder mehrerer natürlicher Programmbefehle des Prozessorkerns ausführt.

16. Vorrichtung nach einem der vorstehenden Ansprüche, wobei der die Sequenz abschließende Befehl ein Mitglied des ersten Satzes von Befehlen ist.

17. Verfahren zum Verarbeiten von Daten unter der Steuerung von Programmbefehlen aus einem ersten Satz von Programmbefehlen oder Programmbefehlen aus einem zweiten Satz von Programmbefehlen, wobei das Verfahren die Schritte aufweist:
Verwendung eines in Software implementierten Befehlsübersetzers, um einen Programmbefehl eines zweiten Satzes von Befehlen als eine Sequenz von Programmbefehlen des ersten Satzes von Programmbefehlen mit einem die Sequenz abschließenden Programmbefehl zu übersetzen, und

in Reaktion auf den Befehl zum Abschließen der Sequenz:
(i) falls eine in Hardware implementierte Befehlsausführungseinheit für den zweiten Satz von Programmbefehlen nicht verfügbar ist, Auslösen einer Übersetzung des nächsten Programmbefehls des zweiten Satzes von Befehlen unter Verwendung des in Software implementierten Befehlsübersetzers, und
(ii) falls die in Hardware implementierte Befehlsausführungseinheit für den zweiten Satz von Programmbefehlen verfügbar ist, Auslösen der Ausführung des nächsten Programmbefehls des zweiten Satzes von Befehlen unter Verwendung der in Hardware implementierten Befehlsausführungseinheit.

18. Verfahren nach Anspruch 17, wobei die Befehlsausführungseinheit in Hardware nicht verfügbar ist, während
(i) die Befehlsausführung in Hardware nicht freigeschaltet ist, oder
(ii) eine Befehlsausführungseinheit in Hardware nicht vorhanden ist.

19. Verfahren nach einem der Ansprüche 17 oder 18, wobei zumindest ein Flag einer Befehlsausführungseinheit in Hardware verwendet wird, um zu erfassen, ob eine Befehlsausführungseinheit in Hardware verfügbar ist oder nicht.

20. Verfahren nach einem der Ansprüche 17, 18 oder 19, wobei der die Sequenz abschließende Befehl eine Startadresse der Übersetzungssoftware innerhalb des in Software implementierten Befehlsübersetzers einer Sequenz von Programmbefehlen des ersten Satzes von Befehlen angibt, welche dazu dienen, den nächsten Programmbefehl des zweiten Satzes von Befehlen zu übersetzen, wobei der in Software implementierte Befehlsübersetzer die Sequenz verwendet, falls die Befehlsausführungseinheit in Hardware nicht verfügbar ist.

21. Verfahren nach Anspruch 20, wobei der die Sequenz abschließende Befehl einen Operanden enthält, der ein Register angibt, welches die Startadresse für die Softwareübersetzung speichert.

22. Verfahren nach Anspruch 18, wobei der die Sequenz abschließende Befehl ein vorbestimmtes Register verwendet, um die Startadresse der Softwareübersetzung zu speichern.

23. Verfahren nach einem der Ansprüche 17 bis 22, wobei der die Sequenz abschließende Befehl eine Startadresse für die Hardwareausführung angibt, welche auf den nächsten Befehl des zweiten Satzes von Befehlen hinweist, wobei die Startadresse für die Hardwareausführung durch die Befehlsausführungseinheit in Hardware verwendet wird, um auf den nächsten Befehl des zweiten Satzes von Befehlen zuzugreifen, falls die Befehlsausführungseinheit in Hardware verfügbar ist.

24. Verfahren nach Anspruch 23, wobei der die Sequenz abschließende Befehl ein vorbestimmtes Register verwendet, um die Startadresse der Hardwareausführung zu speichern.

25. Verfahren nach Anspruch 23, wobei der die Sequenz abschließende Befehl einen Operanden enthält, welcher ein Register angibt, das die Startadresse der in Hardware erfolgenden Ausführung speichert.

26. Verfahren nach einem der Ansprüche 17 bis 25, wobei der Befehlsübersetzer in Software eine Mehrzahl von Sequenzen von Programmbefehlen des ersten Satzes von Befehlen enthält, wobei die Sequenzen dieser Mehrzahl jeweils Programmbefehlen des zweiten Satzes von Befehlen entsprechen.

27. Verfahren nach Anspruch 26, wobei der Befehlsübersetzer in Software eine Tabelle von Zeigern auf die Mehrzahl von Sequenzen enthält.

28. Verfahren nach Anspruch 27, wobei ein Eintrag innerhalb der Tabelle von Zeigern durch einen Programmbefehl des zweiten Satzes von Befehlen, der übersetzt werden soll, indiziert bzw. angezeigt wird.

29. Verfahren nach einem der Ansprüche 27 und 28, wobei eine Basisadresse der Tabelle aus Zeigern innerhalb eines Basisadreßregisters gespeichert ist.

30. Verfahren nach einem der Ansprüche 17 bis 29, welches einen Prozessorkern aufweist, wobei Programmbefehle des ersten Satzes von Befehlen natürliche bzw. inhärente Programmbefehle sind, welche durch den Prozessorkern ausgeführt werden.

31. Verfahren nach Anspruch 30, wobei die Befehlsausführungseinheit in Hardware Befehle in Java-Bytecode als zumindest eine Darstellung eines oder mehrerer natürlicher (inhärenter) Programmbefehle auf dem Prozessorkern ausführt.

32. Verfahren nach einem der Ansprüche 17 bis 31, wobei der die Sequenz abschließende Befehl ein Mitglied des ersten Satzes von Befehlen ist.

33. Computerprogrammprodukt für die Steuerung einer Datenverarbeitungsvorrichtung zur Verarbeitung von Daten unter Steuerung von Programmbefehlen aus einem ersten Satz von Programmbefehlen oder Programmbefehlen aus einem zweiten Satz von Programmbefehlen, wobei das Computerprogrammprodukt aufweist:
eine Befehlsübersetzungslogik in Software, die so betreibbar ist, daß sie einen Programmbefehl des zweiten Satzes von Programmbefehlen als eine Se-

quenz von Programmbefehlen des ersten Satzes von Programmbefehlen übersetzt, die mit einem die Sequenz abschließenden Befehl endet, wobei der die Sequenz abschließende Befehl dazu dient:
(i) falls eine Befehlsausführungseinheit in Hardware für den zweiten Satz von Programmbefehlen nicht verfügbar ist, Auslösen der Übersetzung einer nächsten Programmbefehl des zweiten Satzes von Befehlen unter Verwendung einer Befehlsübersetzungslogik in Software, und
(ii) falls die Befehlsausführungseinheit in Hardware für den zweiten Satz von Befehlen verfügbar ist, Auslösen der Ausführung des nächsten Programmbefehls des zweiten Satzes von Befehlen, unter Verwendung der Befehlsausführungseinheit in Hardware.

34. Computerprogrammprodukt nach Anspruch 33, wobei die Befehlausführungseinheit in Hardware nicht verfügbar ist, während:
(i) die Befehlsausführung in Hardware nicht freigeschaltet ist, oder
(ii) eine Befehlsausführungseinheit in Hardware nicht vorhanden ist.

35. Computerprogrammprodukt nach einem der Ansprüche 33 und 34, wobei zumindest ein Flag einer Befehlsausführungseinheit in Hardware verwendet wird, um zu erfassen, ob eine Befehlsausführungseinheit in Hardware verfügbar ist oder nicht.

36. Computerprogrammprodukt nach einem der Ansprüche 33, 34 und 35, wobei der die Sequenz abschließende Befehl eine Startadresse der Softwareübersetzung innerhalb der Befehlsübersetzungslogik in Software für eine Sequenz von Programmbefehlen des ersten Satzes von Befehlen angibt, die dazu dienen, den nächsten Programmschritt des zweiten Satzes von Befehlen zu übersetzen, wobei die Logik des Befehlsübersetzers in Software die Sequenz verwendet, falls die Befehlsausführungseinheit in Hardware nicht verfügbar ist.

37. Computerprogrammprodukt nach Anspruch 36, wobei der die Sequenz abschließende Befehl einen Operanden enthält, welcher ein Register angibt, das die Startadresse der Softwareübersetzung speichert.

38. Computerprogrammprodukt nach Anspruch 36, wobei der die Sequenz abschließende Befehl ein vorbestimmtes Register verwendet, um die Startadresse für die Softwareübersetzung zu speichern.

39. Computerprogrammprodukt nach einem der Ansprüche 33 bis 38, wobei der die Sequenz abschließende Befehl eine Startadresse für die Ausführung in Hardware angibt, welche auf den nächsten Befehl des zweiten Satzes von Befehlen hinweist, wobei die Startadresse der Hardwareausführung von der Befehlsausführung der Hardware verwendet wird, um auf den nächsten Befehl des zweiten Satzes von Befehlen zuzugreifen, falls die Befehlsausführungseinheit in Hardware verfügbar ist.

40. Computerprogrammprodukt nach Anspruch 39, wobei der die Sequenz abschließende Befehl ein vorbestimmtes Register für die Speicherung der Startadresse für die Ausführung in Hardware verwendet.

41. Computerprogrammprodukt nach Anspruch 39, wobei der die Sequenz abschließende Befehl einen Operanden enthält, welcher ein Register angibt, das die Startadresse für die Hardwareausführung speichert.

42. Computerprogrammprodukt nach einem der Ansprüche 33 bis 41, wobei die Logik für die Befehlsübersetzung in Software eine Mehrzahl von Sequenzen von Programmbefehlen des ersten Satzes von Befehlen enthält, wobei die Sequenzen dieser Mehrzahl jeweils Programmbefehlen des zweiten Satzes von Befehlen entsprechen.

43. Computerprogrammprodukt nach Anspruch 42, wobei die Befehlsübersetzerlogik in Software eine Tabelle von Zeigern auf die Mehrzahl von Sequenzen umfaßt.

44. Computerprogrammprodukt nach Anspruch 43, wobei ein Eintrag innerhalb der Tabelle von Zeigern durch einen Programmbefehl des zweiten Satzes von Befehlen, der übersetzt werden soll, indiziert bzw. angezeigt wird.

45. Computerprogrammprodukt nach einem der Ansprüche 43 oder 44, wobei eine Basisadresse der Tabelle aus Zeigern innerhalb des Basisadreßregisters gespeichert ist.

46. Computerprogrammprodukt nach einem der Ansprüche 33 bis 45, welches einen Prozessorkern aufweist, wobei die Programmbefehle des ersten Satzes von Befehlen natürliche bzw. inhärente Programmbefehle sind, die durch den Prozessorkern ausgeführt werden.

47. Computerprogrammprodukt nach Anspruch 46, wobei die Befehlsausführungseinheit in Hardware Befehle in Java-Bytecode zumindest als eine Wiedergabe einer oder mehrerer natürlicher bzw. ursprünglicher Programmbefehle auf dem Prozessorkern ausführt.

48. Computerprogrammprodukt nach einem der Ansprüche 33 bis 47, wobei der die Sequenz abschließende Befehl ein Mitglied des ersten Satzes von Befehlen ist.

Es folgen 21 Blatt Zeichnungen

Anhängende Zeichnungen

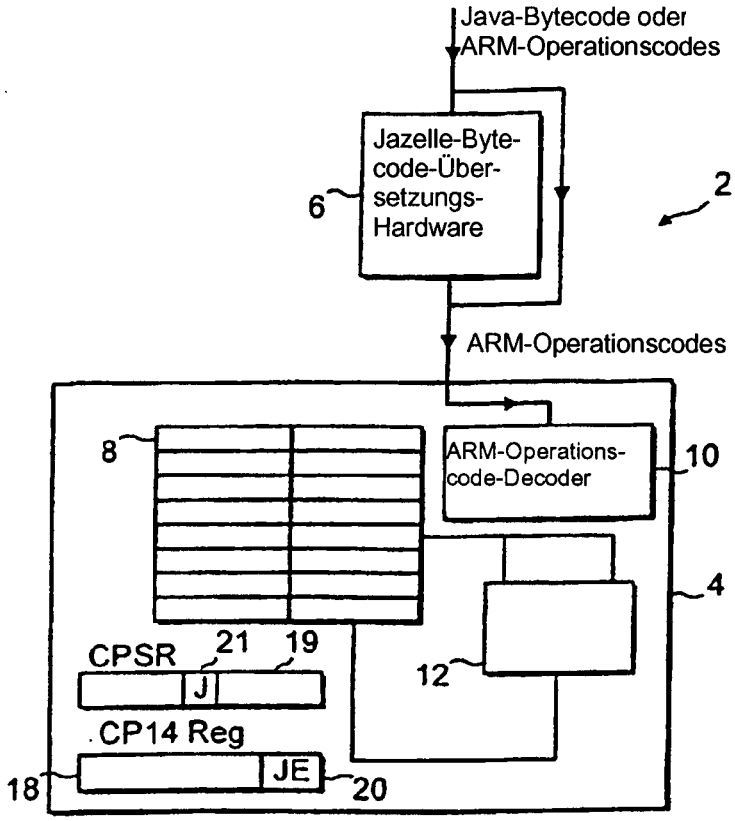

FIG. 1

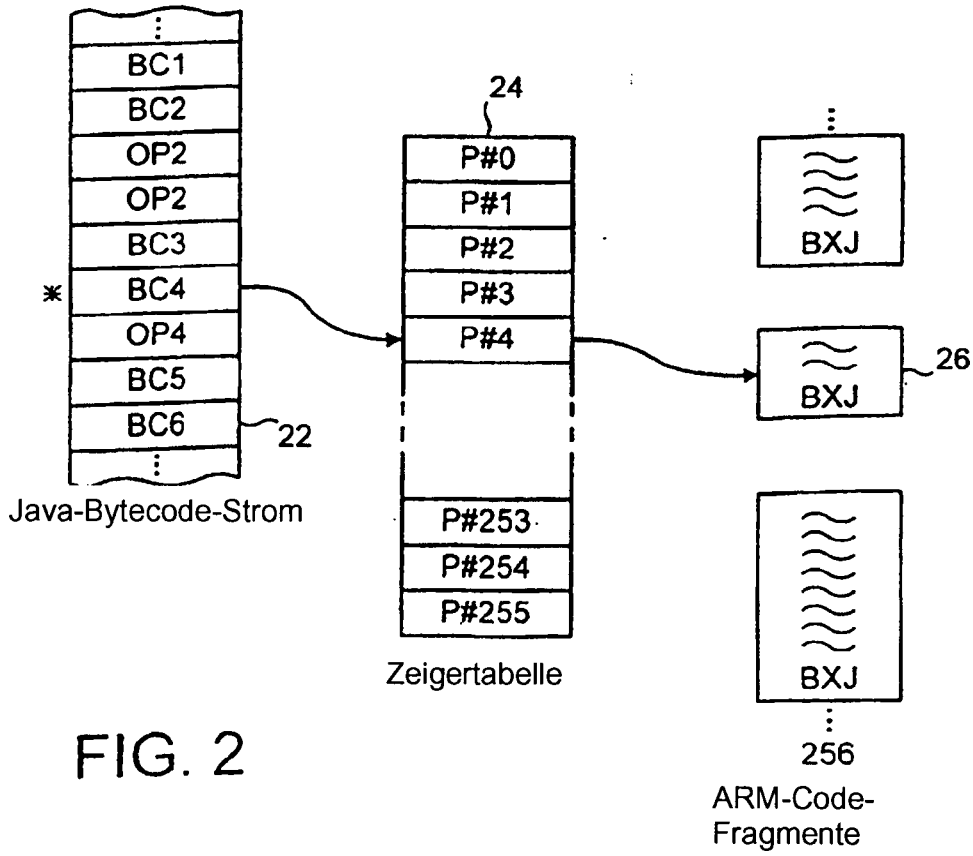

FIG. 2

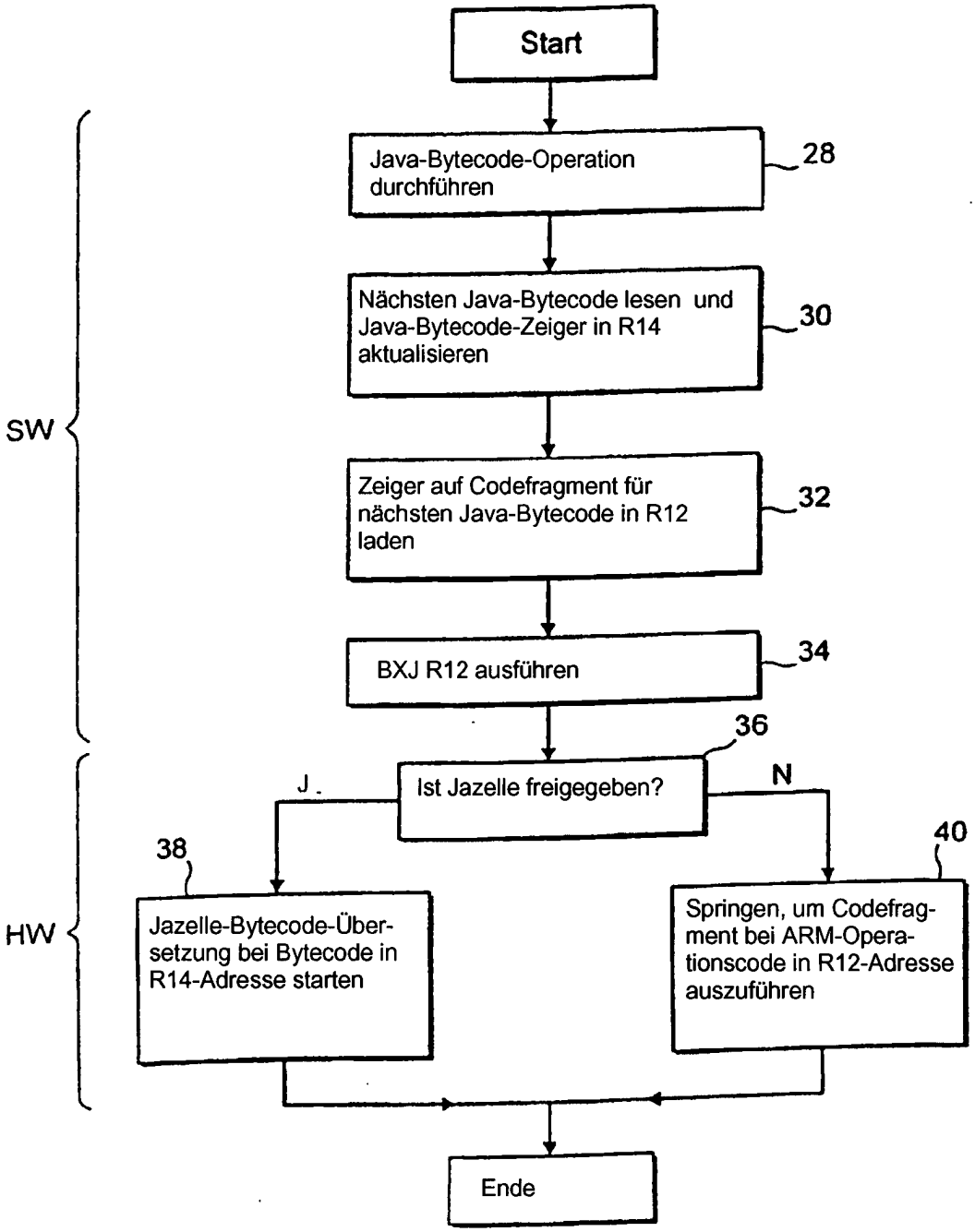
Start
Java-Bytecode-Operation durchführen
28
Nächsten Java-Bytecode lesen und Java-Bytecode-Zeiger in R14 aktualisieren
30
Zeiger auf Codefragment für nächsten Java-Bytecode in R12 laden
32
BXJ R12 ausführen
34
36
Ist Jazelle freigegeben?
J
N
38
Jazelle-Bytecode-Übersetzung bei Bytecode in R14-Adresse starten
40
Springen, um Codefragment bei ARM-Operationscode in R12-Adresse auszuführen
Ende
SW
HW

FIG. 3

```
do_iadd
LDRB    R4, [R14, #1]!   ——— Nächsten Java-Bytecode laden und Bytecode-Zeiger aktualisieren
LDR     R1, [Rstack, #-4]! ——— Ersten Operanden vom Stack holen
LDR     R0, [Rstack, #-4]! ——— Zweiten Operanden vom Stack holen
LDR     R12, [Rexc, R4, LSL #2] ——— Adresse des Codefragments für nächsten Bytecode holen
ADD     R0, R0, R1 ——— Ganzzahl-Addition durchführen
STR     R0, [Rstack], #4 ——— Ergebnis auf Stack legen
BXJ     R12 ——— Nächsten Bytecode in Hardware/Software ausführen
```

FIG. 4

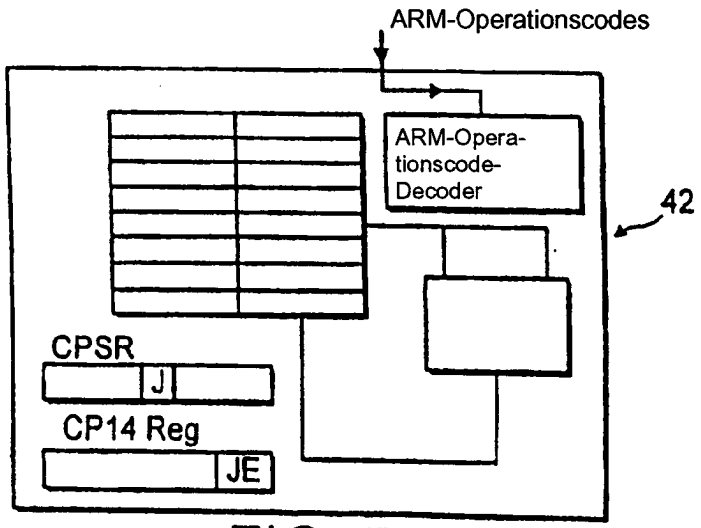

FIG. 5

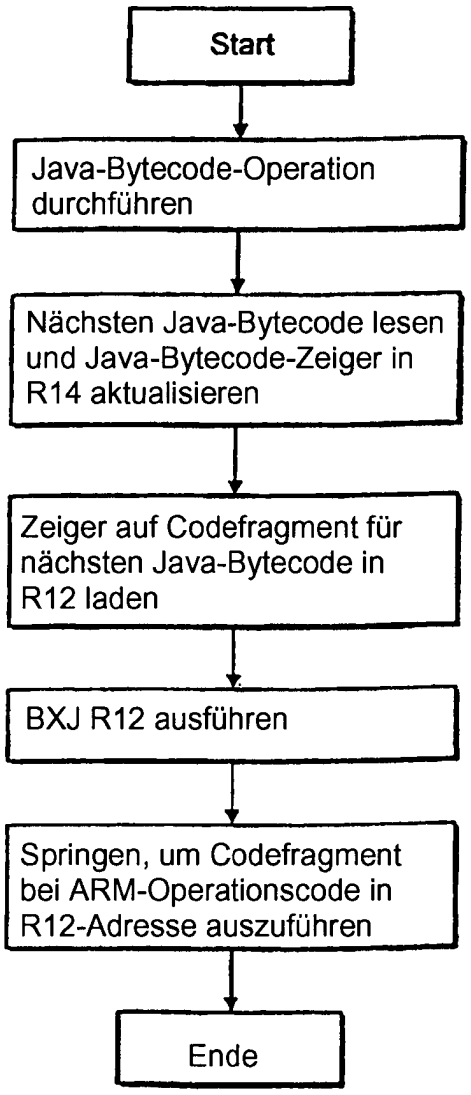
Start
Java-Bytecode-Operation durchführen
Nächsten Java-Bytecode lesen und Java-Bytecode-Zeiger in R14 aktualisieren
Zeiger auf Codefragment für nächsten Java-Bytecode in R12 laden
BXJ R12 ausführen
Springen, um Codefragment bei ARM-Operationscode in R12-Adresse auszuführen
Ende

FIG. 6

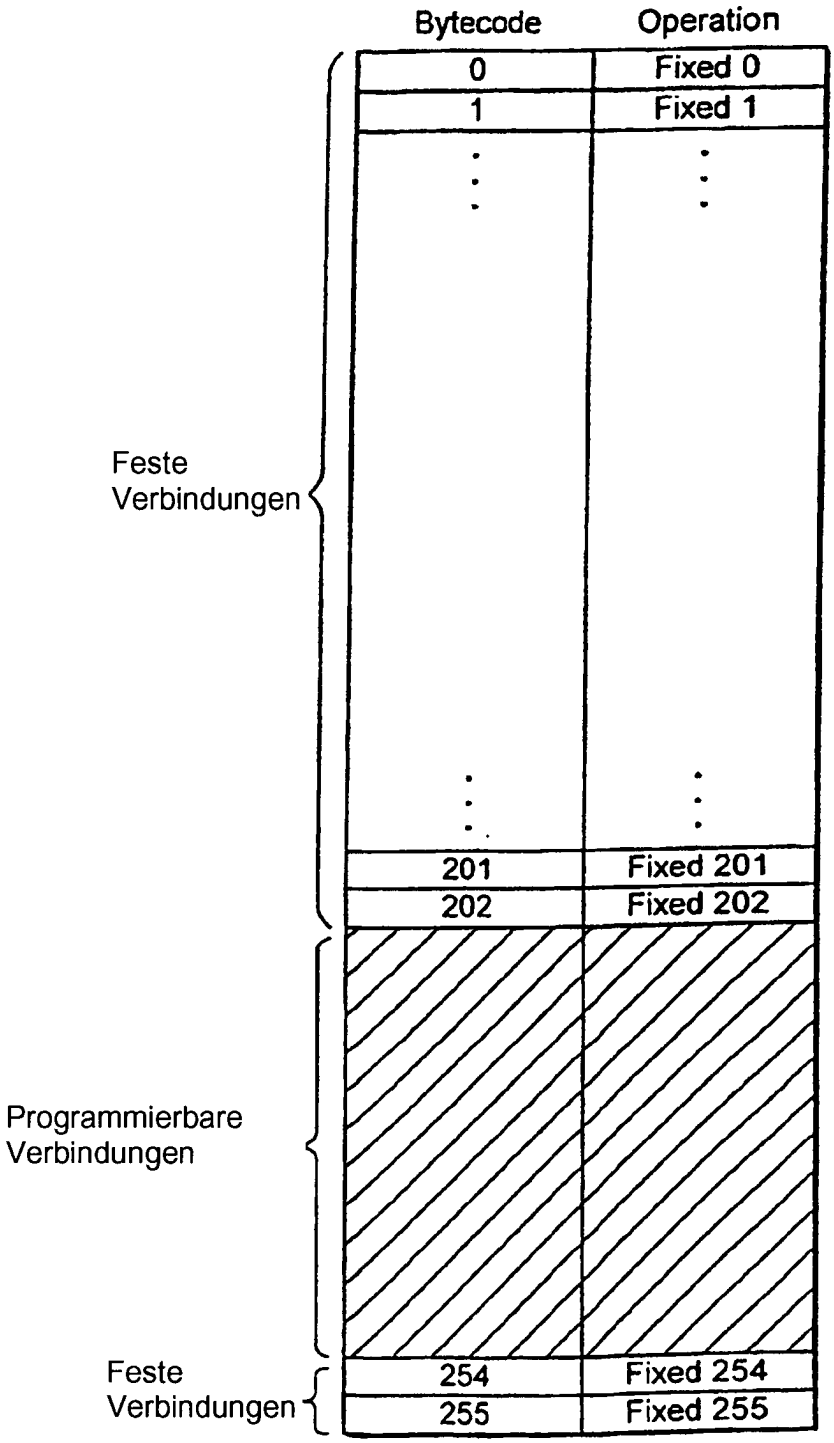
Bytecode
Operation
0
Fixed 0
1
Fixed 1
201
Fixed 201
202
Fixed 202
254
Fixed 254
255
Fixed 255
Feste Verbindungen
Programmierbare Verbindungen
Feste Verbindungen

FIG. 7

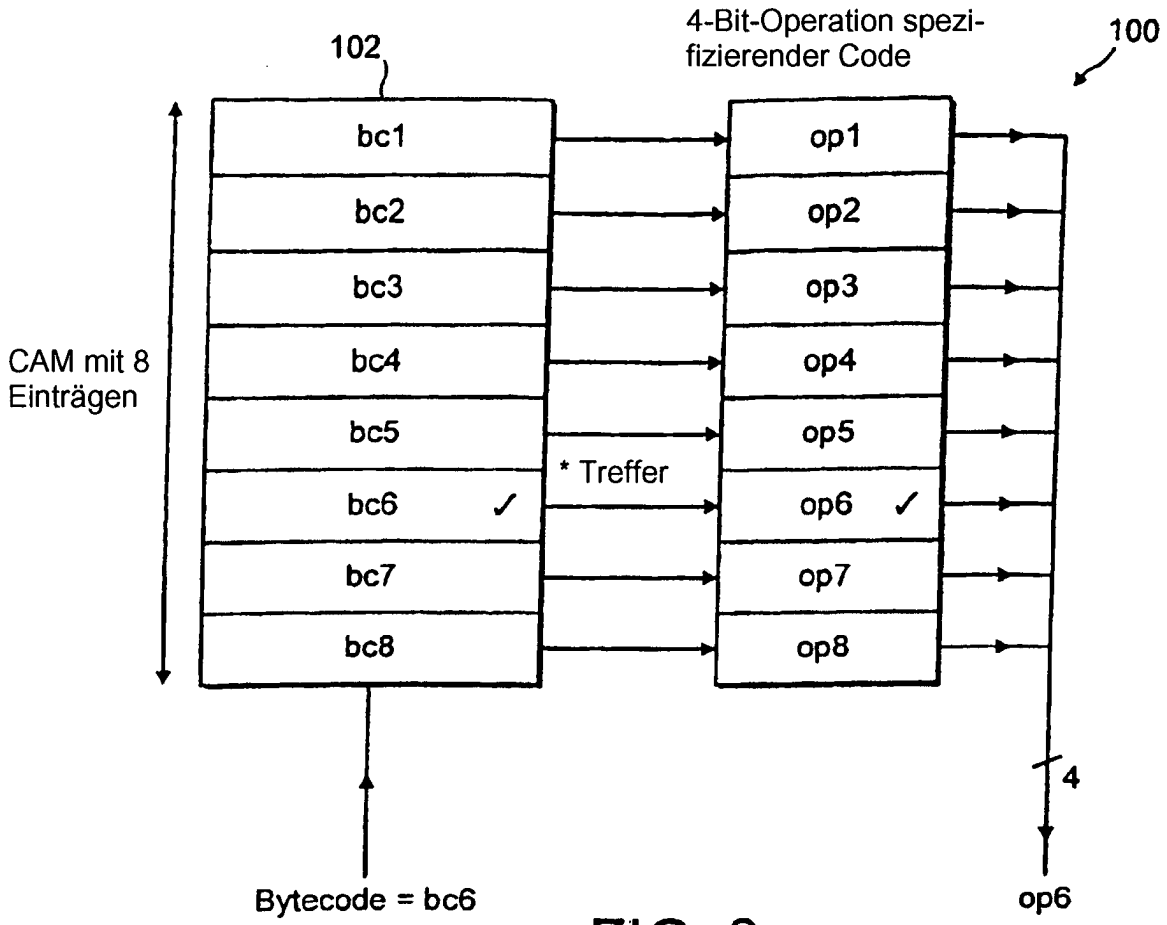
4-Bit-Operation spezi-
fizierender Code
100
102
CAM mit 8
Einträgen
bc1
bc2
bc3
bc4
bc5
bc6
bc7
bc8
* Treffer
op1
op2
op3
op4
op5
op6
op7
op8
4
Bytecode = bc6
op6

FIG. 8

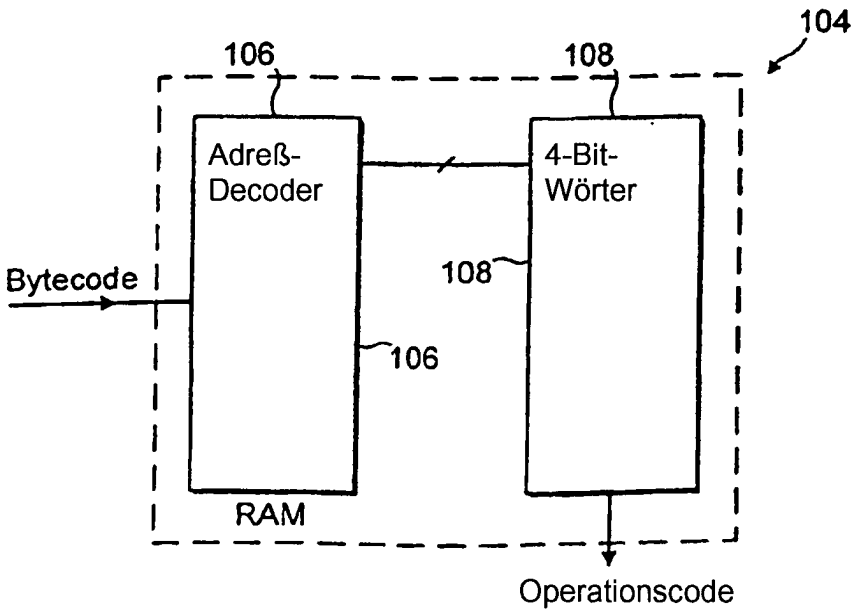
104
106
108
Adreß-
Decoder
4-Bit-
Wörter
Bytecode
108
106
RAM
Operationscode

FIG. 9

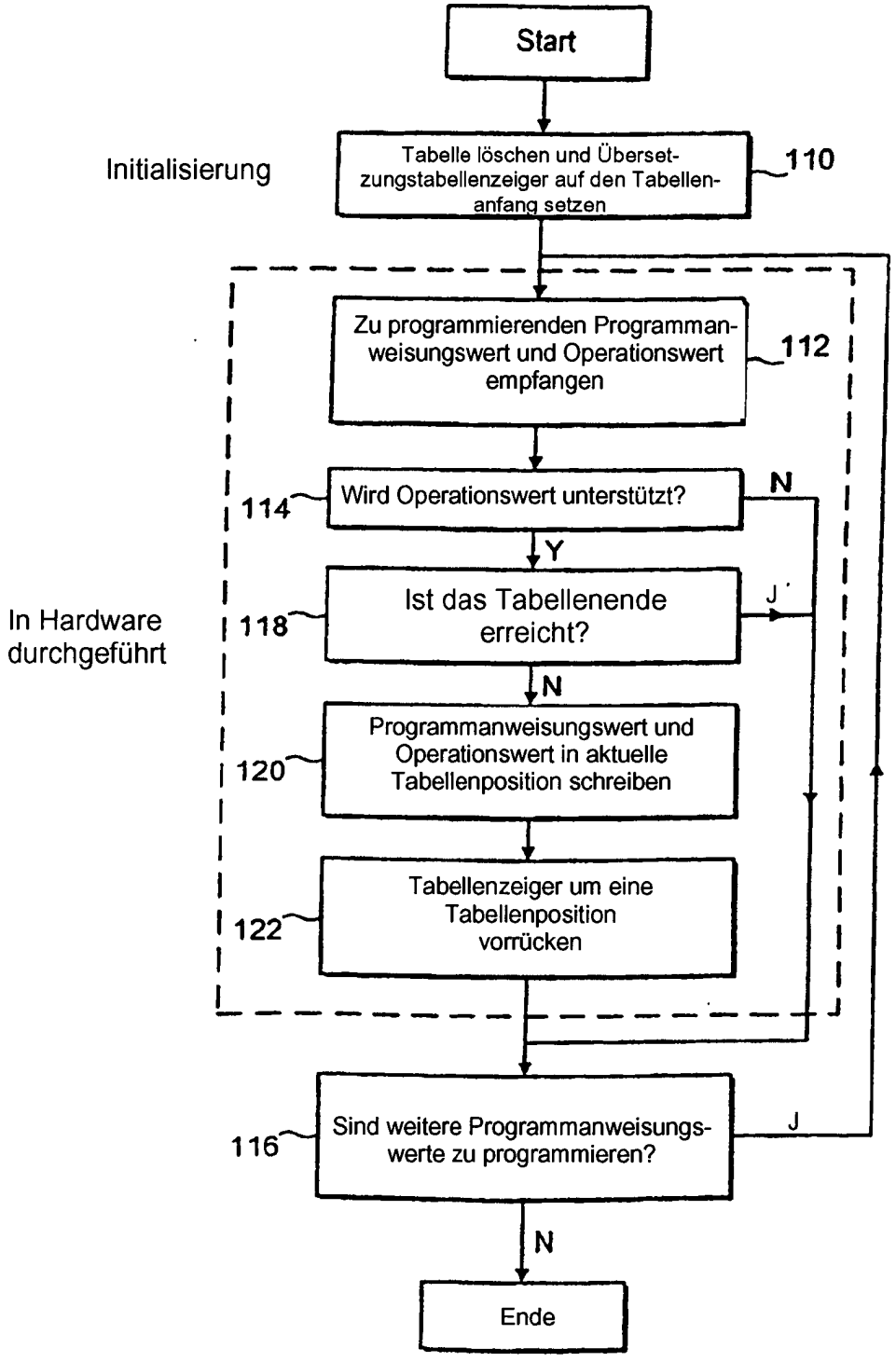
Start
Initialisierung
Tabelle löschen und Übersetzungstabellenzeiger auf den Tabellenanfang setzen
110
Zu programmierenden Programmanweisungswert und Operationswert empfangen
112
114
Wird Operationswert unterstützt?
N
Y
In Hardware durchgeführt
118
Ist das Tabellenende erreicht?
J
N
120
Programmanweisungswert und Operationswert in aktuelle Tabellenposition schreiben
122
Tabellenzeiger um eine Tabellenposition vorrücken
116
Sind weitere Programmanweisungswerte zu programmieren?
J
N
Ende

FIG. 10

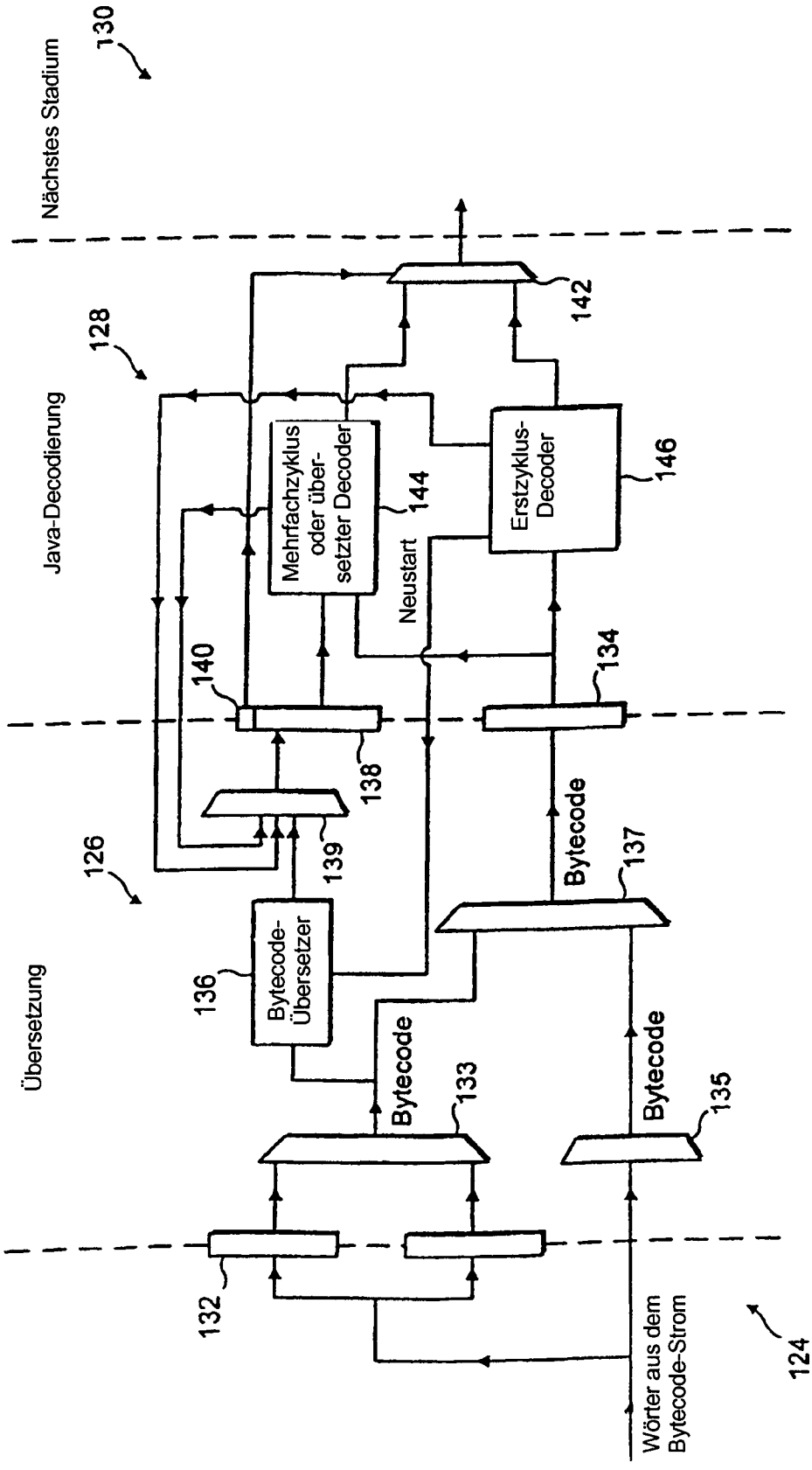
Nächstes Stadium
130
Java-Decodierung
128
Übersetzung
126
Mehrfachzyklus oder übersetzter Decoder
144
Erstzyklus-Decoder
146
Neustart
142
140
138
139
134
Bytecode-Übersetzer
136
Bytecode
133
Bytecode
137
Bytecode
135
132
Wörter aus dem Bytecode-Strom
124

FIG. 11

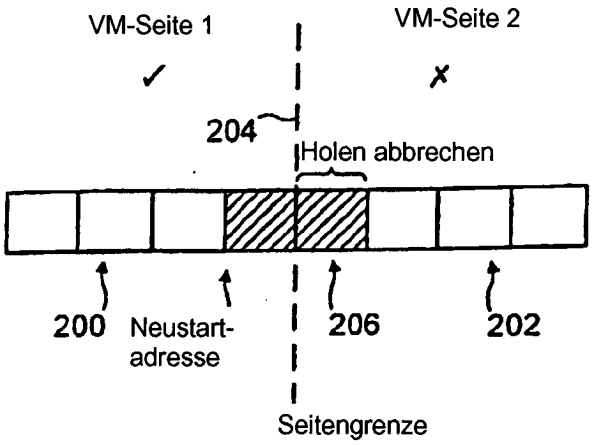

FIG. 12

(PA (Hälfte 1) = Falsch) UND (PA (Hälfte 2) = Wahr

UND

(((Anzahl von Operanden = 1) UND **(bcadd [1:0] = 11))**
ODER ((Anzahl von Operanden = 2) **AND (bcadd [1] = 1)))**

FIG. 14

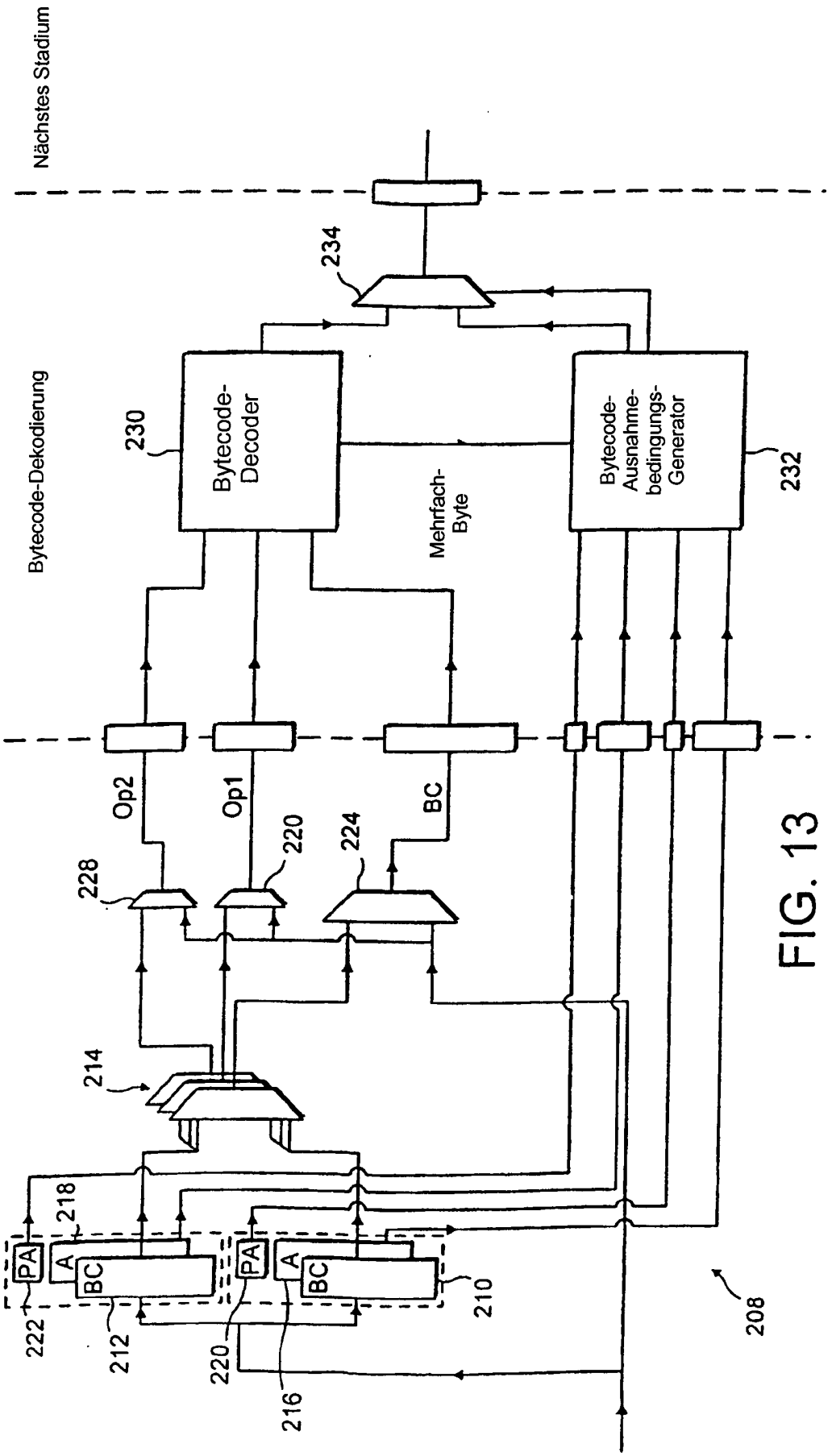
Nächstes Stadium
234
Bytecode-Dekodierung
230
Bytecode-Decoder
Mehrfach-Byte
Bytecode-Ausnahme-bedingungs-Generator
232
Op2
Op1
BC
220
224
228
214
218
PA
A
BC
222
212
220
216
210
208

FIG. 13

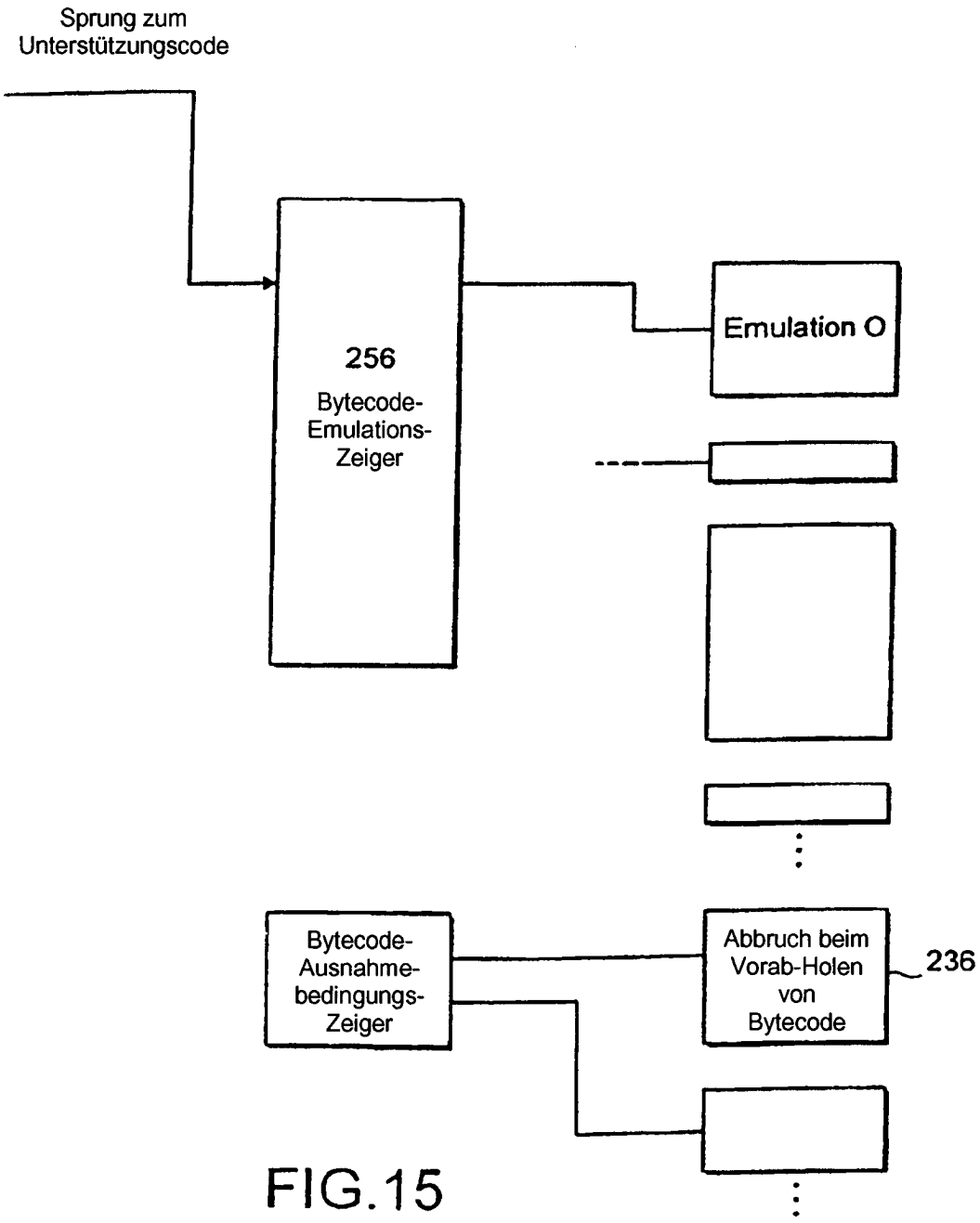
Sprung zum Unterstützungscode
256
Bytecode-Emulations-Zeiger
Emulation O
Bytecode-Ausnahme-bedingungs-Zeiger
Abbruch beim Vorab-Holen von Bytecode
236

FIG.15

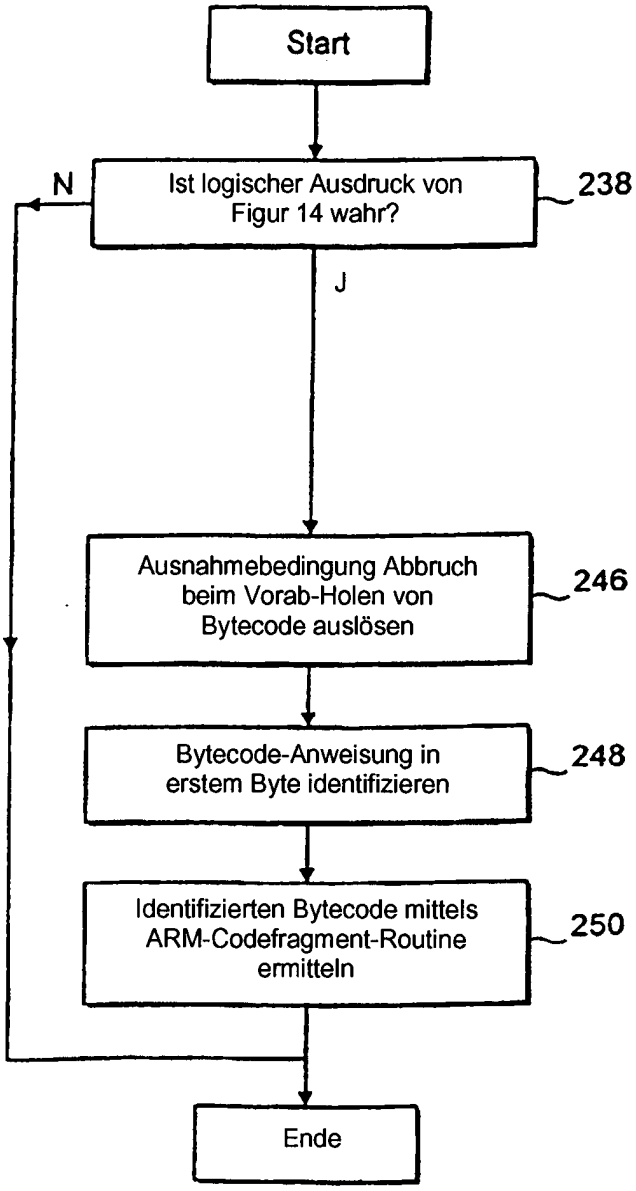
Start
Ist logischer Ausdruck von Figur 14 wahr?
238
N
J
Ausnahmebedingung Abbruch beim Vorab-Holen von Bytecode auslösen
246
Bytecode-Anweisung in erstem Byte identifizieren
248
Identifizierten Bytecode mittels ARM-Codefragment-Routine ermitteln
250
Ende

FIG. 16

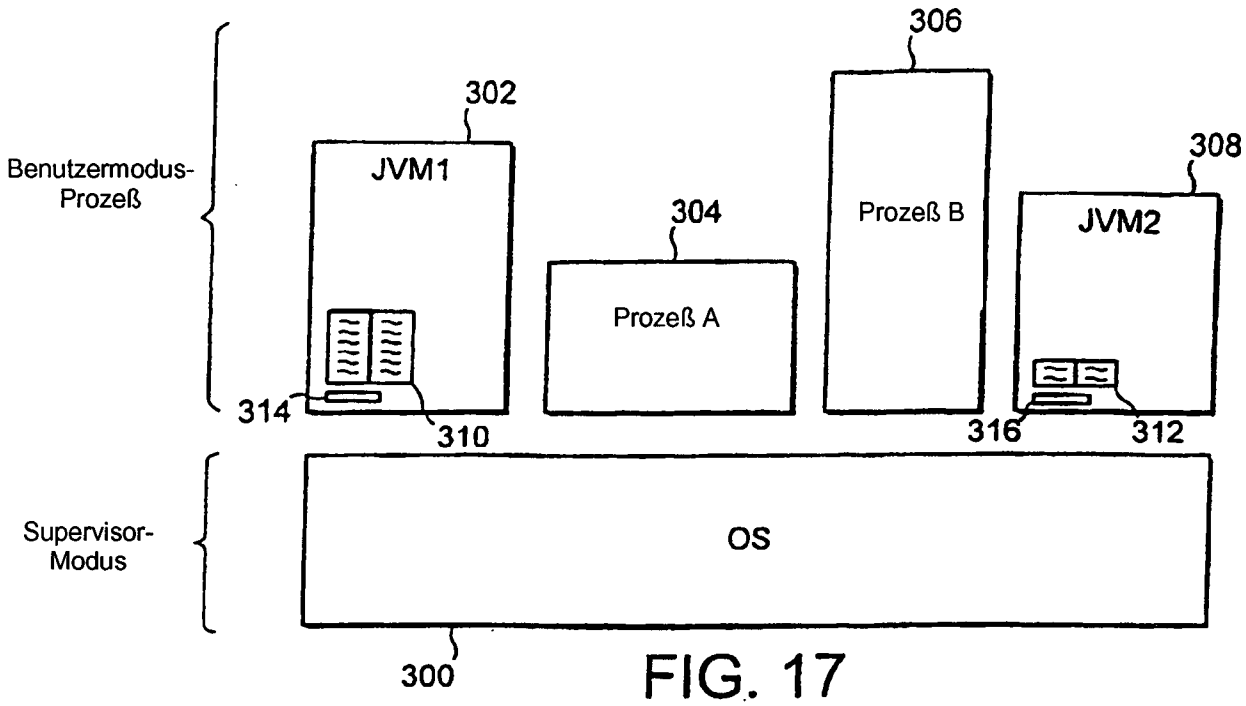
Benutzermodus-Prozeß
Supervisor-Modus
JVM1
Prozeß A
Prozeß B
JVM2
OS
302
304
306
308
310
312
314
316
300

FIG. 17

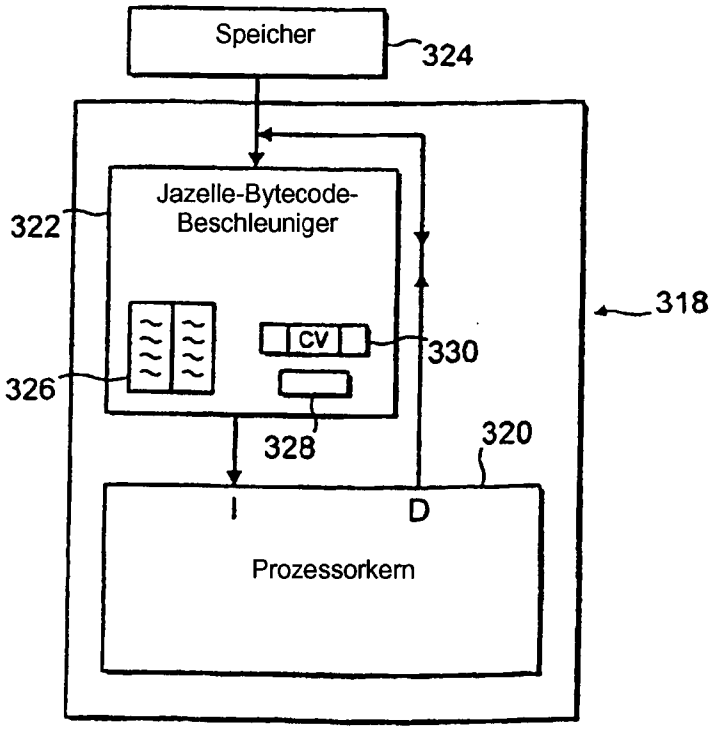
Speicher
324
Jazelle-Bytecode-Beschleuniger
322
CV
330
326
328
320
318
I
D
Prozessorkern

FIG. 18

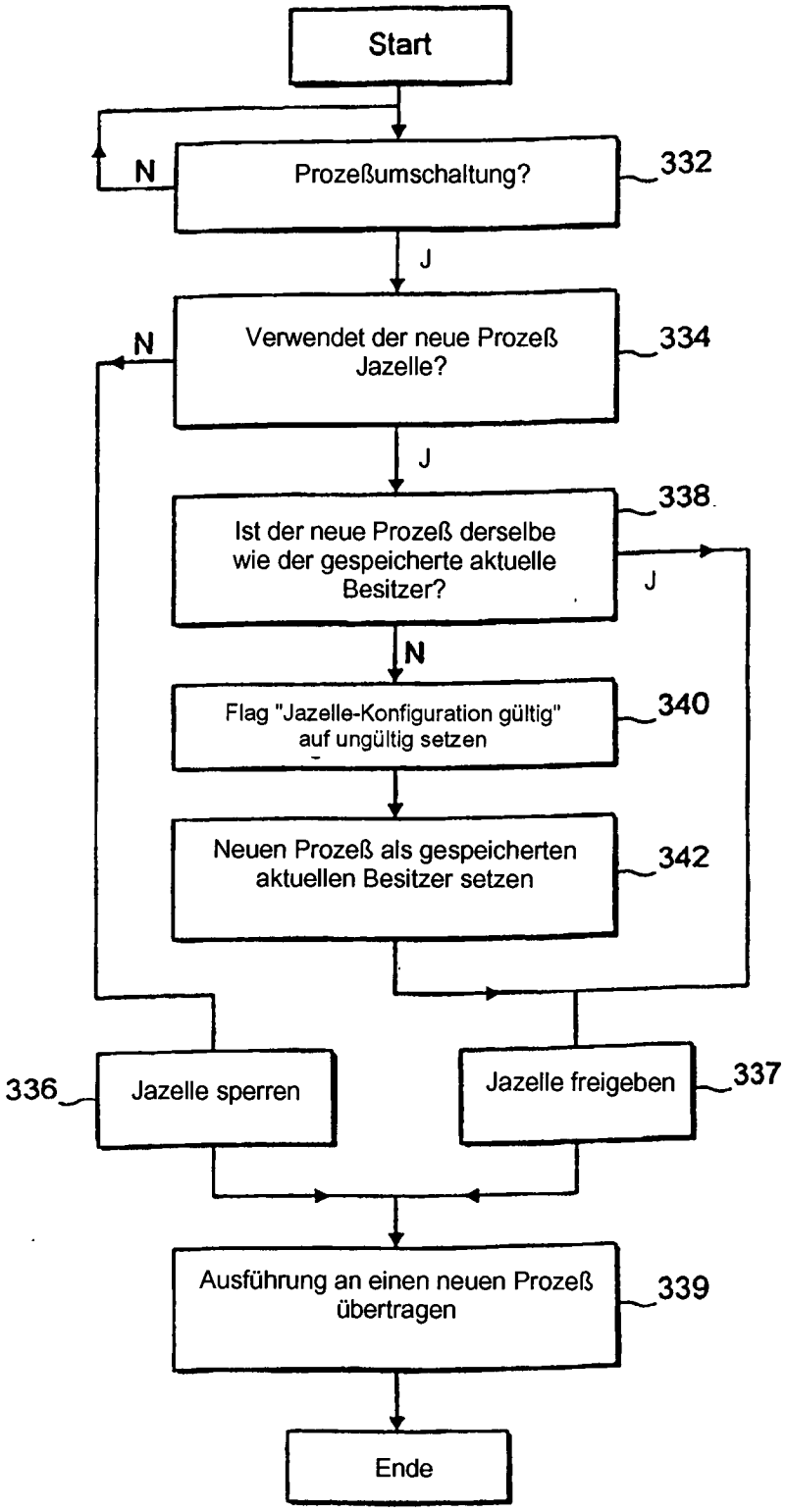
Start
Prozeßumschaltung?
N
J
332
Verwendet der neue Prozeß Jazelle?
N
J
334
Ist der neue Prozeß derselbe wie der gespeicherte aktuelle Besitzer?
J
N
338
Flag "Jazelle-Konfiguration gültig" auf ungültig setzen
340
Neuen Prozeß als gespeicherten aktuellen Besitzer setzen
342
Jazelle sperren
336
Jazelle freigeben
337
Ausführung an einen neuen Prozeß übertragen
339
Ende

FIG. 19

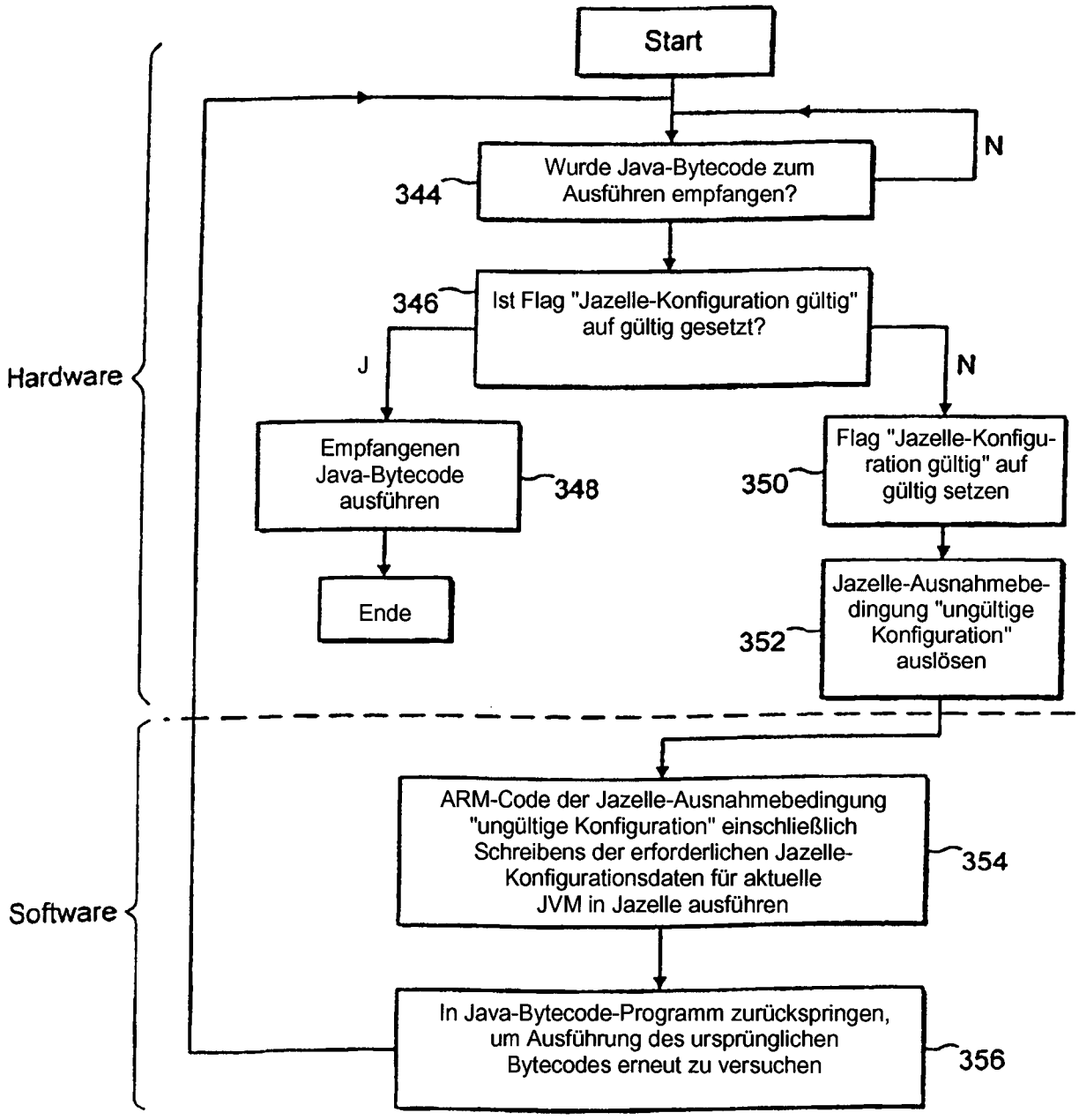
Start
Wurde Java-Bytecode zum Ausführen empfangen?
344
N
Ist Flag "Jazelle-Konfiguration gültig" auf gültig gesetzt?
346
J
N
Hardware
Empfangenen Java-Bytecode ausführen
348
Ende
Flag "Jazelle-Konfiguration gültig" auf gültig setzen
350
Jazelle-Ausnahmebedingung "ungültige Konfiguration" auslösen
352
Software
ARM-Code der Jazelle-Ausnahmebedingung "ungültige Konfiguration" einschließlich Schreibens der erforderlichen Jazelle-Konfigurationsdaten für aktuelle JVM in Jazelle ausführen
354
In Java-Bytecode-Programm zurückspringen, um Ausführung des ursprünglichen Bytecodes erneut zu versuchen
356

FIG. 20

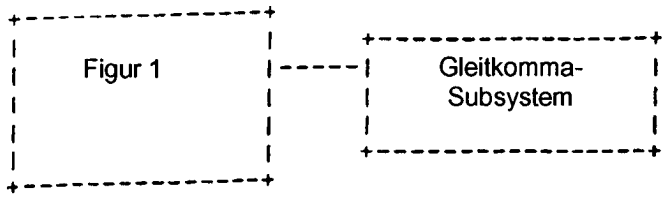

FIG. 21

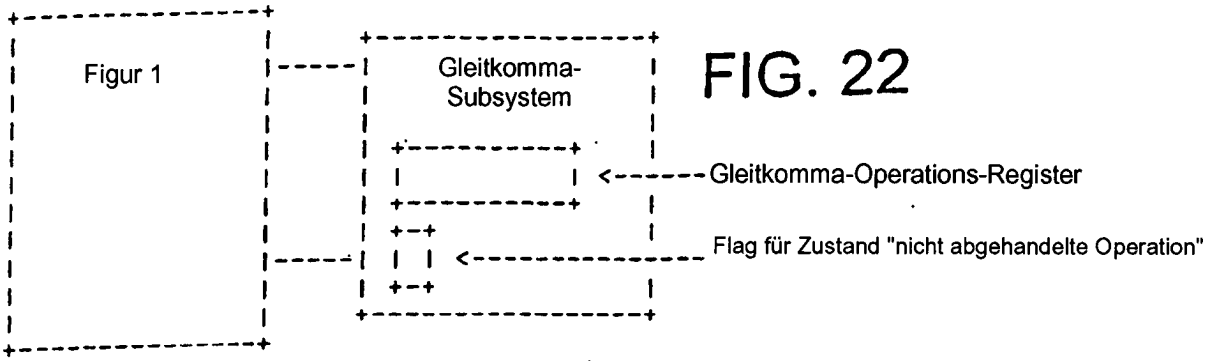

FIG. 22

Einfache Genauigkeit		Doppelte Genauigkeit	
fadd	FADDS Sd, Sn, Sm	dadd	FADDD Dd, Dn, Dm
fsub	FSUBS Sd, Sn, Sm	dsub	FSUBD Dd, Dn, Dm
fmul	FMULS Sd, Sn, Sm	dmul	FMULD Dd, Dn, Dm
fdiv	FDIVS Sd, Sn, Sm	ddiv	FDIVD Dd, Dn, Dm
frem	Not implemented in HW	drem	Nicht in HW implementiert
fneg	FNEGS Sd, Sm	dneg	FNEGD Dd, Dm
f2d	FCVTDS Dd, Sm	d2f	FCVTSD Sd, Dm
f2i	FTOSIZS Sd, Sm	d2i	FTOSIZD Sd, Dm
f2l	Not implemented in HW	d2l	Nicht in HW implementiert
i2f	FSITOS Sd, Sm	i2d	FSITOD Dd, Sm
l2f	Not implemented in HW	l2d	Nicht in HW implementiert
fcmpl	FCMPS/FMSTAT	dcmpl	FCMPD/FMSTAT
fcmpg	FCMPS/FMSTAT	dcmpg	FCMPD/FMSTAT

FIG. 23

```
dmul
        FMULD   D1, D2, D1
dcmpg
        FCMPD   D0, D1
        FMSTAT
        MVNMI   R0, #0
        MOVEQ   R0, #0
        MOVGT   R0, #1
< nächster Java-Bytecode >
```

FIG. 24

```
FMULD   D1, D2, D1          ;Verursacht nicht abgehandelte Operation
FCMPD   D0, D1              ;Nicht abgehandelte Operation wird hier signalisiert
```

Zum ARM-Zustand, um den Bytecode zu emulieren

```
                FMRX Rd, FPSCR  ; Behandlungsroutine für nicht abehandelte
                                ; Operationen führt eine VFP-Anweisung Status-
                                ; Lesen aus, um eine VFP-Ausnahmebedingungs-
                                ; behandlung auszulösen, falls Flag für Zustand
                                ; "nicht abgehandelte Operation" gesetzt ist
```

Flag für Zustand "nicht abgehandelte Operation" ist gesetzt, daher VFP-Ausnahmebedingungsbehandlung durchführen

<Gleitkomma-Operations-Register lesen>
<FMULD tos-1, tos, tos-1 in SW emulieren>
<Flag für Zustand "nicht abgehandelte Operation" löschen>

Zur Behandlungsroutine für nicht abgehandelte Operationen zurückkehren

<Ausführung der Anweisung FMRX Rd, FPSCR erneut versuchen, ohne diesmal eine VFP-Ausnahmebedingungsbehandlung auszulösen >

<Java-Stack in Speicher ausleeren>

```
LDRB    R4, [R14]               ; Bytecode, der nicht abgehandelte Operatior
                                ; auslöste, laden
LDR     R12, [Rexc, R4, LSL #2] ; Adresse des Codefragmente zum Emulierer
                                ; der Anweisung "dcmpg" holen
BX      R12                     ; Zum Codefragment verzweigen
                                ; Beachte, BX statt BXJ zu verwenden, weil
                                ; wir nicht wollen, daß dies in HW ausgeführt
                                ; wird
```

Zum dcmpg Emulationscode verzweigen

```
LDRB    R4, [R14, #1]!          ; Nächsten Java-Bytecode laden und
                                ; Bytecode-Zeiger aktualisieren
FLDD    D1, [Rstack, #-8]!      ; Ersten Operanden vom Stack holen
                                ; 1 Doppeltgenauer Wert = 2 Stackwörter
FLDD    D0, [Rstack, #-8]!      ; Zweiten Operanden vom Stack holen
                                ; Adresse des Codefragmentes für nächsten
LDR     R12, [Rexc, R4, LSL #2] ; Bytecode holen
                                ; Beide doppeltgenauen Werte vergleichen
FCMPD   D0, D1                  ; Ergebnis des Vergleichs lesen
FMSTAT                          ; Ergebnis = -1 wenn <
MVNMI   R0, #0                  ; Ergebnis = 0 wenn =
MOVEQ   R0, #0                  ; Ergebnis = 1 wenn >
MOVGT   R0, #1                  ; Ergebnis auf Stack legen
STR     R0, [Rstack], #4        ; Nächsten Bytecode in Hardware/Software
BXJ     R12                     ; ausführen
```

FIG. 25

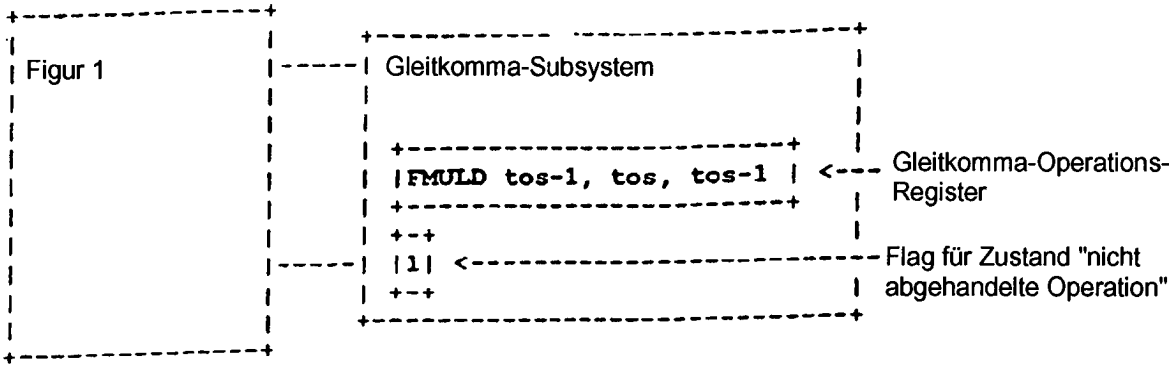
Figur 1
Gleitkomma-Subsystem
FMULD tos-1, tos, tos-1
Gleitkomma-Operations-Register
1
Flag für Zustand "nicht abgehandelte Operation"

FIG. 26

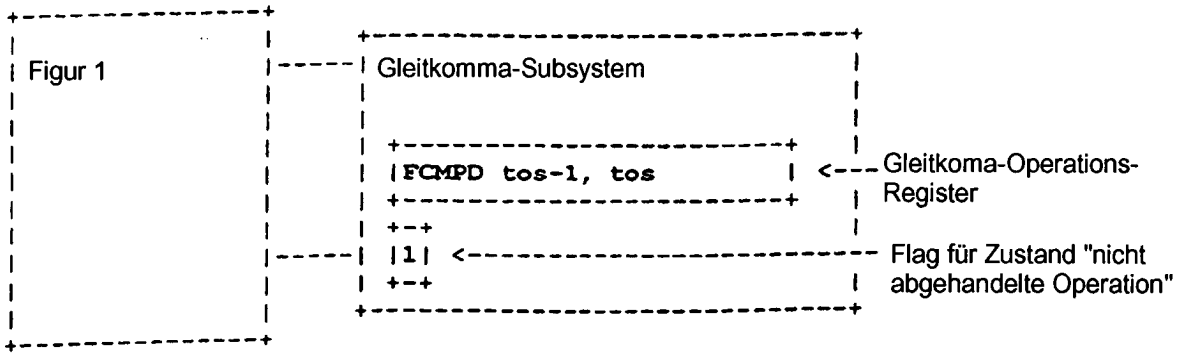
Figur 1
Gleitkomma-Subsystem
FCMPD tos-1, tos
Gleitkoma-Operations-Register
1
Flag für Zustand "nicht abgehandelte Operation"

FIG. 28

```
FMULD   D1, D2, D1          ; Wird normal ausgeführt
FCMPD   D0, D1              ; Verursacht nicht abgehandelte Operation
FMSTAT                      ; Nicht abgehandelte Operation wird hier signalisiert
```

Zum ARM-Zustand, um den Bytecode zu emulieren

```
                FMRX Rd, FPSCR  ; Behandlungsroutine für nicht abgehandelte
                                ; Operationen führt eine VFP-Anweisung Status-
                                ; Lesen aus, um eine VFP-Ausnahmebedingungs-
                                ; behandlung auszulösen, falls Flag für Zustand
                                ; "nicht abgehandelte Operation" gesetzt ist
```

Flag für Zustand "nicht abgehandelte Operation" ist gesetzt, daher VFP-Ausnahmebedingungsbehandlung durchführen

<Gleitkomma-Operations-Register lesen>
<FCMPD tos-1, tos emulieren>
<Flag für Zustand "nicht abgehandelte Operation" löschen>

<Zur Behandlungsroutine für nicht abgehandelte Operationen zurückkehren>

<Ausführung der Anweisung FMRX Rd, FPSCR erneut versuchen, ohne diesmal eine VFP-Ausnahmebedingung auszulösen>

<Java-Stack in Speicher ausleeren>

```
LDRB    R4, [R14]           ; Bytecode, der nicht abgehandelte Operation
                            ; auslöste, laden
LDR     R12, [Rexc, R4, LSL #2] ; Adresse des Codefragmentes zum Emuliere
                            ; der Anweisung "dcmpg" holen
BX      R12                 ; Zum Codefragment verzweigen
                            ; Beachte, BX statt BXJ zu verwenden, weil w
                            ; nicht wollen, daß dies in HW ausgeführt
                            ; wird
```

Zum dcmpg Emulationscode verzweigen

```
LDRB    R4, [R14, #1]!      ; Nächsten Java-Bytecode laden und
                            ; Bytecode-Zeiger aktualisieren
FLDD    D1, [Rstack, #-8]!  ; Ersten Operanden vom Stack holen
                            ; 1 Doppeltgenauer Wert = 2 Stackwörter
FLDD    D0, [Rstack, #-8]!  ; Zweiten Operanden vom Stack holen
                            ; Adresse des Codefragmentes für nächsten
LDR     R12, [Rexc, R4, LSL #2] ; Bytecode holen
                            ; Die Multiplikation erneut versuchen. Dies
FMULD   D0, D0, D1          ; wird auf dasselbe Problem wie zuvor stoßen,
                            ; genau erkannt, jedoch beim Ausführen von
                            ; ARM-Anweisungen statt Bytecodes. Die
                            ; VFP-Ausnahmebedingungsroutine kann dies
                            ; ohne Probleme behandeln
                            ; Ergebnis auf Stack legen, 2 Wörter
FSTD    D0, [Rstack], #8    ; Nächsten Bytecode (dcmpg) in Hardware/
BXJ     R12                 ; Software ausführen
```

FIG. 27

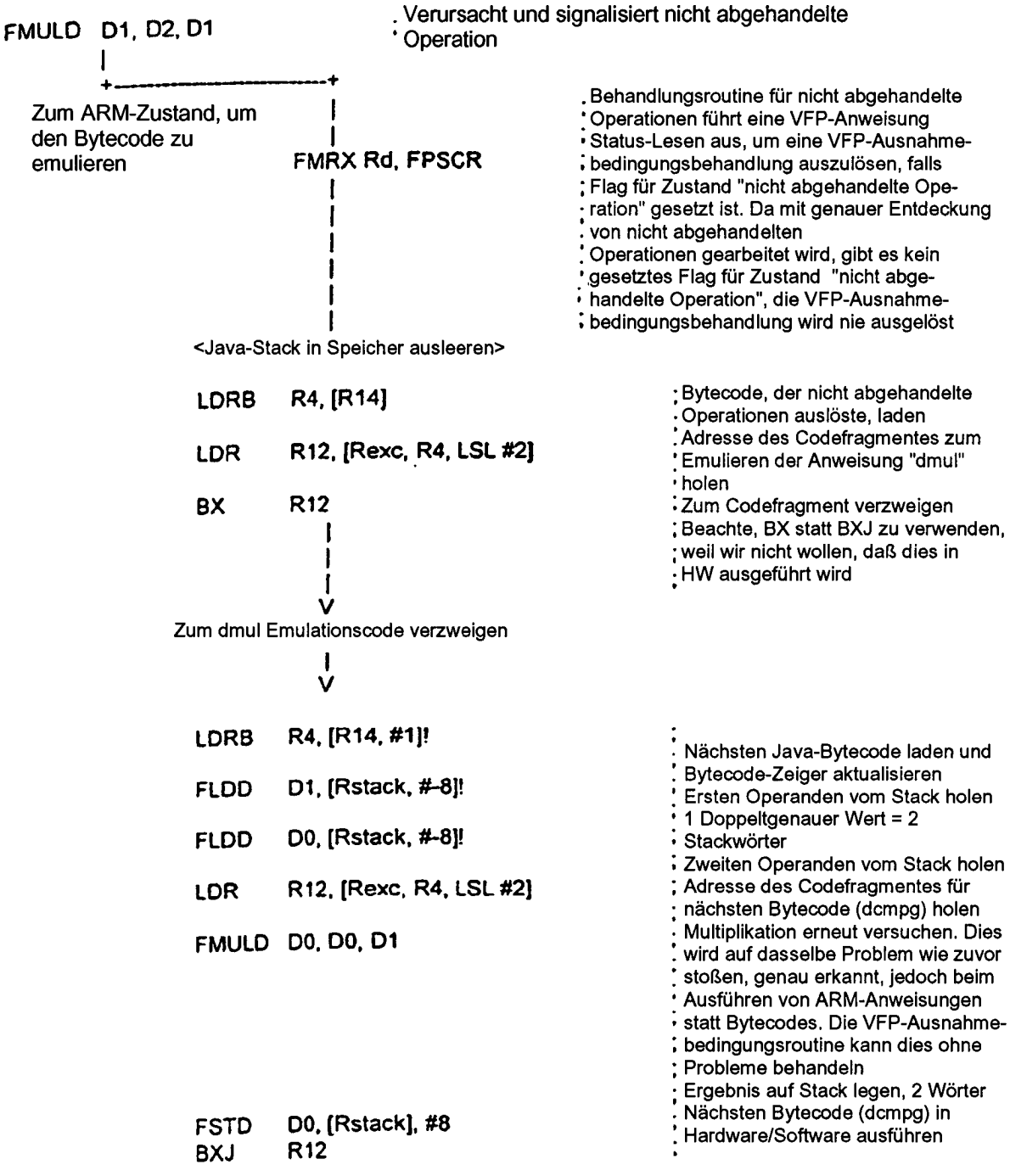
FMULD D1, D2, D1
; Verursacht und signalisiert nicht abgehandelte Operation
Zum ARM-Zustand, um den Bytecode zu emulieren
FMRX Rd, FPSCR
; Behandlungsroutine für nicht abgehandelte Operationen führt eine VFP-Anweisung Status-Lesen aus, um eine VFP-Ausnahmebedingungsbehandlung auszulösen, falls Flag für Zustand "nicht abgehandelte Operation" gesetzt ist. Da mit genauer Entdeckung von nicht abgehandelten Operationen gearbeitet wird, gibt es kein gesetztes Flag für Zustand "nicht abgehandelte Operation", die VFP-Ausnahmebedingungsbehandlung wird nie ausgelöst
<Java-Stack in Speicher ausleeren>
LDRB R4, [R14]
; Bytecode, der nicht abgehandelte Operationen auslöste, laden
LDR R12, [Rexc, R4, LSL #2]
; Adresse des Codefragmentes zum Emulieren der Anweisung "dmul" holen
BX R12
; Zum Codefragment verzweigen
; Beachte, BX statt BXJ zu verwenden, weil wir nicht wollen, daß dies in HW ausgeführt wird
Zum dmul Emulationscode verzweigen
LDRB R4, [R14, #1]!
; Nächsten Java-Bytecode laden und Bytecode-Zeiger aktualisieren
FLDD D1, [Rstack, #-8]!
; Ersten Operanden vom Stack holen
FLDD D0, [Rstack, #-8]!
; 1 Doppeltgenauer Wert = 2 Stackwörter
LDR R12, [Rexc, R4, LSL #2]
; Zweiten Operanden vom Stack holen
; Adresse des Codefragmentes für nächsten Bytecode (dcmpg) holen
FMULD D0, D0, D1
; Multiplikation erneut versuchen. Dies wird auf dasselbe Problem wie zuvor stoßen, genau erkannt, jedoch beim Ausführen von ARM-Anweisungen statt Bytecodes. Die VFP-Ausnahmebedingungsroutine kann dies ohne Probleme behandeln
FSTD D0, [Rstack], #8
; Ergebnis auf Stack legen, 2 Wörter
BXJ R12
; Nächsten Bytecode (dcmpg) in Hardware/Software ausführen

FIG. 29

```
FMULD  D1, D2, D1
FCMPD  D0, D1                    ; Verursacht und signalisiert nicht abgehandelte Operationen
       |
       +-------------------------+
Zum ARM-Zustand, um              |
den Bytecode zu                  |
emulieren                        |
                                 |
                        FMRX Rd, FPSCR   ; Behandlungsroutine für nicht abgehandelte
                                 |       ; Operationen führt eine VFP-Anweisung
                                 |       ; Status-Lesen aus, um eine VFP-Ausnahme-
                                 |       ; bedingungsbehandlung auszulösen, falls
                                 |       ; Flag für Zustand "nicht abgehandelte Operation"
                                 |       ; gesetzt ist. Da mit genauer Entdeckung von nicht
                                 |       ; abgehandelten Operationen gearbeitet wird, gibt
                                 |       ; es kein gesetztes Flag für Zustand "nicht
                                 |       ; abgehandelte Operation", die VFP-Ausnahme-
                                 |       ; bedingungsbehandlung wird nie ausgelöst
<Java-Stack in Speicher ausleeren>

        LDRB    R4, [R14]        ; Bytecode, der nicht abgehandelte Opera-
                                 ; tionen auslöste, laden
        LDR     R12, [Rexc, R4, LSL #2]   ; Adresse des Codefragmentes zum
                                 ; Emulieren der Anweisung "dcmpg"
                                 ; holen
        BX      R12              ; Zum Codefragment verzweigen
                                 ; Beachte, BX statt BXJ zu verwenden, weil
                                 ; wir nicht wollen, daß dies in HW
                                 ; ausgeführt wird
                |
                V
Zum dcmpg Emulationscode verzweigen
                |
                V

        LDRB    R4, [R14, #1]!   ; Nächsten Java-Bytecode laden
                                 ; und Bytecode-Zeiger aktualisieren
        FLDD    D1, [Rstack, #-8]!   ; Ersten Operanden vom Stack
                                 ; holen
        FLDD    D0, [Rstack, #-8]!   ; 1 Doppeltgenauer Wert =
                                 ; 2 Stackwörter
        LDR     R12, [Rexc, R4, LSL #2]   ; Zweiten Operanden vom Stack
                                 ; holen
        FCMPD   D0, D1           ; Adresse des Codefragmentes für
                                 ; nächsten Bytecode holen
                                 ; Vergleich erneut versuchen. Dies
                                 ; wird auf dasselbe Problem wie
                                 ; zuvor stoßen, genau erkannt,
                                 ; jedoch beim Ausführen von
                                 ; ARM-Anweisungen statt
                                 ; Bytecodes. Die VFP-Ausnahme-
                                 ; bedingungsroutine kann dies
                                 ; ohne Probleme behandeln
        FMSTAT                   ; Ergebnis des Vergleichs lesen
        MVNMI   R0, #0           ; Ergebnis = -1 wenn <
        MOVEQ   R0, #0           ; Ergebnis = 0 wenn =
        MOVGT   R0, #1           ; Ergebnis = 1 wenn >
        STR     R0, [Rstack], #4 ; Ergebnis auf Stack legen
        BXJ     R12              ; Nächsten Bytecode in Hardware/
                                 ; Software ausführen
```

FIG. 30