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

(10) **DE 600 38 693 T2** 2009.07.02

(12) # Übersetzung der europäischen Patentschrift

(97) **EP 1 236 107 B1**
(21) Deutsches Aktenzeichen: **600 38 693.7**
(86) PCT-Aktenzeichen: **PCT/US00/28213**
(96) Europäisches Aktenzeichen: **00 970 828.0**
(87) PCT-Veröffentlichungs-Nr.: **WO 2001/041529**
(86) PCT-Anmeldetag: **11.10.2000**
(87) Veröffentlichungstag der PCT-Anmeldung: **14.06.2001**
(97) Erstveröffentlichung durch das EPA: **04.09.2002**
(97) Veröffentlichungstag der Patenterteilung beim EPA: **23.04.2008**
(47) Veröffentlichungstag im Patentblatt: **02.07.2009**

(51) Int Cl.[8]: ***G06F 9/46*** (2006.01)

(30) Unionspriorität:
458589 **09.12.1999** **US**

(84) Benannte Vertragsstaaten:
AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LI, LU, MC, NL, PT, SE

(73) Patentinhaber:
Intel Corporation, Santa Clara, Calif., US

(74) Vertreter:
Hauck Patent- und Rechtsanwälte, 80339 München

(72) Erfinder:
RODGERS, Dion, Hillsboro, OR 97123, US; TOLL, Bret, Hillsboro, OR 97124, US; WOOD, Amiee, Hillsboro, OR 97124, US

(54) Bezeichnung: **VERFAHREN UND VORRICHTUNG ZUR AUSSCHALTUNG EINES TAKTSIGNALS IN EINEM VIELFADENPROZESSOR**

Beschreibung

GEBIET DER TECHNIK

[0001] Die vorliegende Erfindung betrifft allgemein das Gebiet der multithreading-fähigen Prozessoren und im Besonderen ein Verfahren und eine Vorrichtung zur Deaktivierung eines Taktsignals in einem multithreading-fähigen (MT) Prozessor.

STAND DER TECHNIK

[0002] Die Bauweise der multithreading-fähigen (MT) Prozessoren gilt seit einiger Zeit als eine zunehmend attraktive Option für die Steigerung der Performance von Prozessoren. Das Multithreading in einem Prozessor stellt unter anderem das Potenzial für eine effektivere Nutzung verschiedener Prozessorressourcen bereit und im Besonderen für eine effektivere Nutzung der Ausführungslogik in einem Prozessor. Im Besonderen können durch die Zufuhr bzw. Einspeisung mehrerer Threads in die Ausführungslogik eines Prozessors Taktzyklen, die im anderen Fall ungenutzt wären aufgrund eines Stopps oder eine andere Verzögerung bei der Verarbeitung eines bestimmten Threads, für die Behandlung eines weiteren Threads genutzt werden. Ein Anhalten bzw. ein Stopp der Verarbeitung eines bestimmten Threads kann die Folge einer Reihe verschiedener Ereignisse in einer Prozessor-Pipeline sein. Zum Beispiel führt ein Cache Miss oder eine flache Verzweigungsvorhersage (d. h. eine Operation mit langer Latenzzeit) für einen Befehl, die in einem Thread enthalten ist, für gewöhnlich dazu, dass die Verarbeitung des relevanten Threads angehalten bzw. unterbrochen wird. Der negative Effekt von Operationen mit langer Latenzzeit auf die Effizienz der Ausführungslogik wird verstärkt durch die letzten Steigerungen des Durchsatzes von Ausführungslogiken, welche über die Fortschritte in Bezug auf die Speicherzugriffs- und Speicherabrufraten hinausgehen.

[0003] Multithreading-fähige Computeranwendungen werden ferner zunehmend zur Regel in Anbetracht der für diese multithreading-fähigen Anwendungen bereitgestellten Unterstützung durch eine Reihe beliebter Betriebssysteme, wie zum Beispiel die Betriebssysteme Windows NT® und Unix. Multithreading-fähige Computeranwendungen sind im Bereich multimedialer Umgebungen besonders effizient.

[0004] Multithreading-fähige Prozessoren können weit gefasst gemäß des eingesetzten Thread-Interleavingmusters oder der Schalt- bzw. der eingesetzten Switching Methode in dem relevanten Prozessor in zwei Kategorien eingeteilt werden (d. h. in feine oder grobe Konstruktionen bzw. Designs). Feine multithreading-fähige Konstruktionen unterstützen mehrere aktive Threads in einem Prozessor und verschachteln für gewöhnlich zwei unterschiedliche Threads von Zyklus zu Zyklus. Grobe multithreading-fähige Konstruktionen verschachteln für gewöhnlich die Anweisungen bzw. Befehle verschiedener Threads beim Auftreten eines Ereignisses mit langer Latenzzeit, wie zum Beispiel bei einem Cache Miss. In „Evaluation of Multithreaded Uniprocessors for Commercial Application Environments" von R. Eickemayer, R. Johnson, et al., The 23rd Annual International Symposium an Computer Architecture, Seiten 203–212, Mai 1996, wird ein grobes multithreading-fähiges Design beschrieben. Die Unterscheidungen zwischen feinen und groben Designs bzw. Konstruktionen werden näher erläutert in „Architectural and Implementation Tradeoffs in the Design of Multiple-Context Processors" von J. Laudon, A. Gupta, Multithreaded Computer Architectures: A Summary of the State of the Art, herausgegeben bzw. editiert von R. A. Iannuci et al., Seiten 167–200, Kluwer Academic Publishers, Norwell, Massachusetts, USA, 1994. Laudeon schlägt ferner eine Interleaving-Methode vor, welche das Schalten von Zyklus zu Zyklus einer feinen Konstruktion mit den vollständigen Pipeline-Interlocks eines groben Designs (oder blockierten Systems) kombiniert. Zu diesem Zweck schlägt Laudon einen Befehl „Back off" vor, der einen spezifischen Thread (oder Kontext) für eine bestimmte Anzahl von Zyklen nicht verfügbar macht. Ein derartiger Befehl „Back off" kann beim Auftreten vorbestimmter Ereignisse ausgegeben werden, wie etwa bei einem Cache Miss. Auf diese Weise vermeidet Laudon die Notwendigkeit für die Durchführung eines tatsächlichen Thread-Wechsels bzw. Thread-Umschaltens, indem einer der Threads einfach nicht verfügbar gemacht wird.

[0005] Das U.S. Patent US-A-5.392.437 offenbart einen Mechanismus zum Abschalten einer funktionalen Einheit einer integrierten Schaltung mit mehreren funktionalen Einheiten, die durch unabhängige Takte getaktet werden, transparent und unabhängig von den restlichen funktionalen Einheiten, und zwar indem die funktionale Einheit mit Informationen versehen wird, die angeben, ob sie verwendet werden muss, und wobei die funktionale Einheit ab- bzw. ausgeschaltet wird, wenn sie nicht verwendet werden muss. Der unabhängige Takt der funktionalen Einheit wird angehalten, wenn die funktionale Einheit nicht verwendet wird, und er wird automatisch gestartet, wenn die funktionale Einheit nicht verwendet werden muss.

[0006] Eine multithreading-fähige Architektur für einen Prozessor ist in Verbindung mit einer spekulativen Out-of-Order-Ausführungsprozessorarchitektur mit einer Reihe von Herausforderungen verbunden. Im Besonderen gestaltet sich die Behandlung von Ereignissen (z. B. Verzweigungsbefehle, Ausnahmen oder Unterbrechungen), die zu einer unerwarteten Veränderung des Flusses eines Befehlsstroms füh-

ren können, komplizierter, wenn mehrere Threads berücksichtigt werden müssen. In einem Prozessor, in dem eine gemeinsame Nutzung von Ressourcen zwischen mehreren Threads implementiert wird (d. h. es gibt eine begrenzte oder keine Duplizierung von funktionalen Einheiten für jeden von dem Prozessor unterstützten Thread), gestaltet sich die Behandlung des Auftretens von Ereignissen, welche einen bestimmten Thread betreffen, dahingehend komplizierter, dass weitere Threads bei der Behandlung derartiger Ereignisse berücksichtigt werden müssen.

[0007] Wenn eine gemeinsame Nutzung von Ressourcen (englisch: Resource Sharing) in einem multithreading-fähigen Prozessor eingesetzt wird, so ist es ferner wünschenswert, eine erhöhte Nutzung der gemeinsam genutzten Ressourcen zu versuchen, als Reaktion auf Veränderungen des Zustands der in dem multithreading-fähigen Prozessor bearbeiteten Threads.

ZUSAMMENFASSUNG DER ERFINDUNG

[0008] Vorgesehen ist gemäß der vorliegenden Erfindung ein Verfahren, welches das Verwalten einer Anzeige eines anstehenden Ereignisses in Bezug auf mehrere Threads, die in einem multithreading-fähigen Prozessor unterstützt werden, aufweist. Verwaltet wird eine Anzeige eines anstehenden Ereignisses in Bezug auf mehrere Threads, die in einem multithreading-fähigen Prozessor unterstützt werden. Detektiert wird eins Taktdeaktivierungszustand, der durch die Anzeige von keinen anstehenden Ereignissen in Bezug auf jeden der mehreren Threads angezeigt wird, sowie ein inaktiver Zustand für jeden der mehreren Threads. Ein Taktsignal, sofern freigegeben, wird deaktiviert in Bezug auf mindestens eine funktionale Einheit in dem multithreading-fähigen Prozessor als Reaktion auf das Detektieren des Taktdeaktivierungszustands.

[0009] Weitere Merkmale der vorliegenden Erfindung werden aus den beigefügten Zeichnungen und aus der folgenden genauen Beschreibung deutlich.

KURZE BESCHREIBUNG DER ZEICHNUNGEN

[0010] Die vorliegende Erfindung ist in den Abbildungen der beigefügten Zeichnungen beispielhaft und ohne einzuschränken veranschaulicht. in den Zeichnungen sind übereinstimmende Elemente mit den gleichen Bezugszeichen bezeichnet. In den Zeichnungen zeigen:

[0011] Fig. 1 ein Blockdiagramm eines Ausführungsbeispiels einer Pipeline eines Prozessors mit Multithreading-Unterstützung;

[0012] Fig. 2 ein Blockdiagramm eines exemplarischen Ausführungsbeispiels eines Prozessors in Form eines universellen multithreading-fähigen Mikroprozessors;

[0013] Fig. 3 ein Blockdiagramm ausgewählter Komponenten eines exemplarischen multithreading-fähigen Mikroprozessors und im Besonderen verschiedener funktionaler Einheiten, die eine Pufferkapazität (oder Speicherkapazität) so logisch partitioniert bereitstellt, dass mehrere Threads berücksichtigt werden können;

[0014] Fig. 4 ein Blockdiagramm eines Out-of-Order-Clusters gemäß einem Ausführungsbeispiel;

[0015] Fig. 5 eine schematische Darstellung einer Register-Alias-Tabelle und einer Registerdatei sowie gemäß der Verwendung in einem Ausführungsbeispiel;

[0016] **Fig.** 6 ein Blockdiagramm von Einzelheiten in Bezug auf einen Neuordnungspuffer gemäß einem Ausführungsbeispiel, der logisch so aufgeteilt ist, dass er mehrere Threads in einem multithreading-fähigen Prozessors bearbeitet;

[0017] Fig. 6B eine schematische Darstellung eines Registers für ein anstehendes bzw. ausstehendes Ereignis und eines Ereignissperrregisters gemäß einem Ausführungsbeispiel;

[0018] Fig. 7A ein Flussdiagramm eines Verfahrens gemäß einem Ausführungsbeispiel zur Verarbeitung eines Ereignisses in einem multithreading-fähigen Prozessor;

[0019] Fig. 7B ein Flussdiagramm eines Verfahrens gemäß einem Ausführungsbeispiel zur Behandlung eines Ereignisses „virtual Nuke" in einem multithreading-fähigen Prozessor;

[0020] Fig. 8 eine schematische Darstellung einer Reihe exemplarischer Ereignisse, die von einem Ereignisdetektor detektiert werden können, implementiert in einem multithreading-fähigen Prozessor, gemäß einem Ausführungsbeispiel;

[0021] die Fig. 9 und Fig. 10 entsprechende Blockdiagramme von beispielhaftem Inhalt einer Neuordnungstabelle in einem exemplarischen Neuordnungspuffer, wie dieser etwa in Fig. 6A dargestellt ist;

[0022] Fig. 11A ein Flussdiagramm eines Verfahrens gemäß einem exemplarischen Ausführungsbeispiel zur Ausführung einer Löschoperation (Nuke) in einem multithreading-fähigen Prozessor, der mindestens erste und zweite Threads unterstützt;

[0023] Fig. 11B ein Blockdiagramm einer Konfigurationslogik gemäß einem exemplarischen Ausfüh-

rungsbeispiel, wobei die Logik so arbeitet, dass eine funktionale Einheit gemäß der Ausgabe einer aktiven Thread-Zustandsmaschine konfiguriert wird;

[0024] Fig. 12 ein Zeitsteuerungsdiagramm der Aktivierung eines Nuke-Signals gemäß einem Ausführungsbeispiel;

[0025] Fig. 13 ein Flussdiagramm eines Verfahrens gemäß einem Ausführungsbeispiel der Bereitstellung eines exklusiven Zugriffs auf eine Ereignisbehandlungsroutine in einem multithreading-fähigen Prozessor;

[0026] Fig. 14 ein Zustandsdiagramm, das gemäß einem Ausführungsbeispiel die Operation einer exklusiven Zugriffs-Zustandsmaschine implementiert in einem multithreading-fähigen Prozessor darstellt;

[0027] Fig. 15 ein Zustandsdiagramm, das gemäß einem Ausführungsbeispiel Zustände veranschaulicht, die von einer aktiven Thread-Zustandsmaschine belegt werden können, die in einem multithreading-fähigen Prozessor implementiert ist;

[0028] Fig. 16A ein Flussdiagramm, das ein Verfahren gemäß einem Ausführungsbeispiel veranschaulicht zum Verlassen eines aktiven Threads bei der Erkennung eines Ruheereignisses für den aktiven Thread in einem multithreading-fähigen Prozessor;

[0029] Fig. 16B eine schematische Darstellung des Speichers des Zustands und der Delokation von Registern beim Verlassen eines Threads gemäß einem Ausführungsbeispiel;

[0030] Fig. 17 ein Flussdiagramm eines Verfahrens gemäß einem Ausführungsbeispiels des Übergangs eines Threads aus einem inaktiven Zustand in einen aktiven Zustand beim Erkennen eines Unterbrechungsereignisses für den inaktiven Thread;

[0031] Fig. 18 ein Flussdiagramm eines Verfahrens gemäß einem Ausführungsbeispiel des Verwaltens der Aktivierung und der Deaktivierung eines Taktsignals an mindestens eine funktionale Einheit in einem multithreading-fähigen Prozessor;

[0032] Fig. 19A ein Blockdiagramm der Taktsteuerlogik gemäß einem Ausführungsbeispiel zum Aktivieren und Deaktivieren eines Taktsignals in einem multithreading-fähigen Prozessor; und

[0033] Fig. 19B eine Prinzipskizze eines Ausführungsbeispiels der Taktsteuerlogik aus Fig. 19A.

GENAUE BESCHREIBUNG

[0034] Beschrieben werden ein Verfahren und eine Vorrichtung zur Verwaltung eines Taktsignals in einem multithreading-fähigen Prozessor. In der folgenden Beschreibung sind zu Zwecken der Erläuterung zahlreiche besondere Einzelheiten ausgeführt, um ein umfassendes Verständnis der vorliegenden Erfindung zu vermitteln. Für den Fachmann auf dem Gebiet der Erfindung ist es jedoch ersichtlich, dass die vorliegende Erfindung auch ohne diese besonderen Einzelheiten ausgeführt werden kann.

[0035] Für die Zwecke der vorliegenden Patentschrift beinhaltet der Begriff „Ereignis" jedes Ereignis, in einem Prozessor sowie außerhalb eines Prozessors, das eine Veränderung oder Unterbrechung der Be- bzw. Verarbeitung eines Befehlsstroms (Makro- oder Mikrobefehle) in einem Prozessor bewirkt. Folglich umfasst der Begriff „Ereignis" unter anderem auch die Verarbeitungen von Verzweigungsbefehlen, Ausnahmen und Unterbrechungen, die in dem Prozessor oder außerhalb des Prozessors erzeugt werden können.

[0036] Für die Zwecke der vorliegenden Patentschrift bezieht sich der Begriff „Prozessor" auf jede Maschine bzw. Vorrichtung, die in der Lage ist, eine Sequenz bzw. Folge von Befehlen (z. B. Makro- oder Mikrobefehle) zu verarbeiten und umfasst unter anderem universelle Mikroprozessoren, Mikroprozessoren für einen bestimmten Zweck, Grafiksteuereinheiten, Audiosteuereinheiten, Multimedia-Steuereinheiten, Mikrocontroller oder Netzwerksteuereinheiten. Ferner bezieht sich der Begriff „Prozessor" ferner auf CISC-Prozessoren (Complex Instruction Set Computer-Prozessoren), RISC-Prozessoren (Reduced Instruction Set Computer-Prozessoren) bzw. VLIW-Prozessoren (Very Long Instruction Word-Prozessoren).

[0037] Ferner umfasst der Begriff „Auslösepunkt" (Clearing Point) alle in einem Befehlsstrom (einschließlich eines Mikrobefehls- oder Makrobefehlsstroms) durch eine Flussmarkierung oder einen anderen Befehl enthaltenen Befehle einer Position in dem Befehlsstrom, an der ein Ereignis behandelt oder verarbeitet werden kann.

[0038] Der Begriff „Anweisung" bzw. „Befehl" umfasst unter anderem und ohne einzuschränken einen Makrobefehl oder einen Mikrobefehl.

[0039] Bestimmte exemplarische Ausführungsbeispiele der vorliegenden Erfindung werden mit einer primären Implementierung in Hardware oder Software beschrieben. Der Fachmann auf dem Gebiet wird aber erkennen, dass zahlreiche Merkmale leicht auch in Hardware, Software oder einer Kombination aus Hardware und Software implementiert werden können. Die Software (z. B. entweder Mikrobefehle oder Makrobefehle) für die Implementierung von Ausführungsbeispielen der vorliegenden Erfindung können sich vollständig oder zumindest teilweise in

einem Hauptspeicher befinden, auf den ein Prozessor zugreifen kann, wobei sich die Software aber auch in dem Prozessor selbst befinden kann (z. B. in einem Cache oder einem Mikrocode-Sequencer bzw. einer Mikrocode-Ablaufsteuerung). Zum Beispiel können die Ereignisbehandlungsroutinen und Zustandsmaschinen in Mikrocode, getrennt von einem Mikrocode-Sequencer implementiert werden.

[0040] Software kann ferner über eine Netzwerkschnittstellenvorrichtung übertragen oder empfangen werden.

[0041] Für die Zwecke der vorliegenden Patentschrift umfasst der Begriff „maschinenlesbares Medium" jedes Medium, das eine Folge von Befehlen für die Ausführung durch die Maschine bzw. Vorrichtung speichern oder codieren kann und das bewirkt, dass die Maschine eine beliebige der Methodologien der vorliegenden Erfindung ausführt. Der Begriff „maschinenlesbares Medium" beinhaltet somit unter anderem und ohne darauf beschränkt zu sein Festkörperspeicher, optische und magnetische Plattenspeicher und Trägerwellensignale.

Prozessor-Pipeline

[0042] Die Abbildung aus Fig. 1 zeigt ein Blockdiagramm auf höherer Ebene und veranschaulicht ein Ausführungsbeispiel der Prozessor-Pipeline **10**. Die Pipeline **10** weist eine Reihe von Pipe-Stufen auf, beginnend mit einer Erfassungs-Pipe-Stufe **12**, in der Befehle (z. B. Makrobefehle) abgerufen und der Pipeline **10** zugeführt werden. Zum Beispiel kann ein Makrobefehl aus einem Cache-Speicher abgerufen werden, der integral mit dem Prozessor ist oder diesem eng zugeordnet ist, wobei ein Makrobefehl aber auch über einen Prozessorbus aus einem externen Hauptspeicher abgerufen werden kann. Von der Erfassungs-Pipe-Stufe **12** werden die Makrobefehle zu einer Decodierungs-Pipe-Stufe **14** ausgebreitet, wo Makrobefehle in Mikrobefehle (auch als „Mikrocode" bezeichnet) übersetzt werden, die sich zur Ausführung in dem Prozessor eignen. Die Mikrobefehle werden danach weiter entlang dem Pfad zu einer Zuordnungs-Pipe-Stufe **16** ausgebreitet, wo die Prozessorressourcen den verschiedenen Mikrobefehlen gemäß der Verfügbarkeit und den Anforderungen zugeordnet werden. Die Mikrobefehle werden danach in einer Ausführungsstufe **18** ausgeführt, bevor sie in einer Rückzugs-Pipe-Stufe **20** zurückgezogen oder „zurückgeschrieben" (z. B. in einen Architekturzustand versetzt) werden.

Mikroprozessorarchitektur

[0043] Die Abbildung aus Fig. 2 zeigt ein Blockdiagramm eines exemplarischen Ausführungsbeispiels eines Prozessors **30** in Form eines Universal-Mikroprozessors. Der Prozessor **30** wird nachstehend als ein multithreading-fähiger (MT) Prozessor beschrieben und ist demgemäß in der Lage, mehrere Befehls-Threads (oder Kontexte) zu verarbeiten. Eine Reihe der nachstehend in der Patentschrift bereitgestellten Lehren ist jedoch nicht spezifisch für einen multithreading-fähigen Prozessor, und wobei diese Lehren auch in einem Prozessor mit einem Thread Anwendung finden können. In einem exemplarischen Ausführungsbeispiel kann der Prozessor **30** einen Mikroprozessor mit Intel-Architektur (IA) umfassen, der den Intel Architecture Befehlssatz ausführen kann. Ein Beispiel für einen derartigen Mikroprozessor mit Intel-Architektur ist der Mikroprozessor Pentium Pro® oder der Mikroprozessor Pentium III®, hergestellt von der Intel Corporation, Santa Clara, Kalifornien, USA.

[0044] In einem Ausführungsbeispiel umfasst der Prozessor **30** ein In-Order-Front-End und ein Out-of-Order-Back-End. Das In-Order-Front-End weist eine Busschnittstelleneinheit **32** auf, die als Leitung zwischen dem Prozessor **30** und anderen Komponenten (z. B. Hauptspeicher) eines Computersystems fungiert, in dem der Prozessor **30** eingesetzt werden kann. Zu diesem Zweck koppelt die Busschnittstelleneinheit **32** den Prozessor **30** mit einem Prozessorbus (nicht abgebildet), über den Daten und Steuerinformationen empfangen werden können an dem Prozessor **30** und von diesem aus ausgebreitet werden können. Die Busschnittstelleneinheit **32** weist eine Front Side Bus (FSB) Logik **34** auf, welche die Kommunikationen über den Prozessorbus steuert. Die Busschnittstelleneinheit **32** weist ferner eine Buswarteschlange **36** auf, welche eine Pufferfunktion bereitstellt in Bezug auf Kommunikationen über den Prozessorbus. Gemäß der Abbildung empfängt di Busschnittstelleneinheit **32** Busanforderungen **38** von einer Speicherausführungseinheit **42** und sendet Snoops oder Busantworten an diese Einheit zurück, welche eine lokale Speicherkapazität in dem Prozessor **30** bereitstellt. Die Speicherausführungseinheit **42** weist einen Unified Daten- und Befehls-Cache **44**, einen Daten Translation Lookaside Buffer bzw. Adressumsetzpuffer (TLB) **46** und einen Speicherordnungspuffer **48** auf. Die Speicherausführungseinheit **42** empfängt Befehlserfassungsanforderungen **50** von einer Mikrobefehls-Translation-Engine **54** und liefert unverarbeitete Befehle **52** (d. h. codierte Makrobefehle) an die Mikrobefehls-Translation-Engine **54**, welche die empfangenen Makrobefehle in einen entsprechenden Satz bzw. Vorrat von Mikrobefehlen übersetzt.

[0045] Die Mikrobefehls-Translation-Engine **54** arbeitet effektiv als eine Trache-Cache-Miss-Behandlungsroutine, wobei sie im Falle eines Trache-Cache-Misses Mikrobefehle einem Trace-Cache **62** bereitstellt. Zu diesem Zweck arbeitet die Mikrobefehls-Translation-Engine **54** so, dass sie die Erfassungs- und Decodierungs-Pipe-Stufen **12** und **14** für

den Fall des Auftretens eines Trace-Cache-Misses bereitstellt. Die Mikrobefehls-Translation-Engine **54** umfasst gemäß der Abbildung einen Zeiger auf den nächsten Befehl (NIP) **100**, einen Befehls-Adressumsetzpuffer (TLB) **102**, einen Verzweigungsvorhersageeinrichtung **104**, einen Befehls-Streaming-Puffer **106**, einen Befehlsvordecodierer **108**, eine Befehlsleitlogik **110**, einen Befehlsdecodierer **112** und eine Verzweigungsadressberechnungseinheit **114**. Der Zeiger auf den nächsten Befehl **100**, der TLB **102**, der Verzweigungsvorhersageeinrichtung **104** und der Befehls-Streaming-Puffer **106** bildet gemeinsam eine Verzweigungsvorhersageeinheit bzw. Branch Prediction Unit (BPU) **99**. Der Befehlsdecodierer **1122** und die Verzweigungsadressberechnungseinheit **114** umfassen gemeinsam eine Befehlsumsetzungseinheit (IX-Einheit) **113**.

[0046] Der Zeiger auf den nächsten Befehl **100** gibt Anforderungen für einen nächsten Befehl an den Unified Cache **44** aus. In dem exemplarischen Ausführungsbeispiel, in dem der Prozessor **30** einen multithreading-fähigen Prozessor umfasst, der zwei Threads verarbeiten kann, kann der Zeiger auf den nächsten Befehl **100** einen Multiplexer (MUX) (nicht abgebildet) aufweisen, der zwischen Befehlszeigern auswählt, die entweder dem ersten oder dem zweiten Thread zugeordnet sind, in Bezug auf die Integration in die nächste ausgegebene Befehlsanforderung. In einem Ausführungsbeispiel verschachtelt der Zeiger auf den nächsten Befehl **100** Anforderungen für einen nächsten Befehl für die ersten und zweiten Threads Zyklus für Zyklus („Ping Pong"), wobei angenommen wird, dass Befehle für beide Threads angefordert worden sind, und wobei die Ressourcen des Befehls-Streaming-Puffers **106** für beide Threads nicht erschöpft sind. Die Anforderungen des Zeigers auf den nächsten Befehl können auf 16, 32 oder 64 Bytes lauten, abhängig davon, ob sich die Adresse der ursprünglichen Anforderung in der oberen Hälfte einer auf 32 Byte oder 64 Byte ausgerichteten Zeile befinden. Der Zeiger auf den nächsten Befehl **100** kann durch die Verzweigungsvorhersageeinrichtung **104**, die Verzweigungsadressberechnungseinheit **114** oder den Trace-Cache **62** umgeleitet werden, wobei eine Trace-Cache-Miss-Anforderung die Umleitungsanforderung mit der höchsten Priorität darstellt.

[0047] Wenn der Zeiger auf den nächsten Befehl **100** von dem Unified Cache **44** einen Befehl anfordert, erzeugt er einen "Anforderungsbezeichner" von zwei Bit, der der Befehlsanforderung zugeordnet ist und als ein "Tag" (Kennzeichen) für die relevante Befehlsanforderung dient. Wenn als Reaktion auf eine Befehlsanforderung Daten zurückgegeben werden, gibt der Unified Cache **44** die folgenden Tags oder Bezeichner gemeinsam mit den Daten zurück:

1. Den „Anforderungsbezeichner", der von dem Zeiger auf den nächsten Befehl **100** vorgesehen wird;
2. Einen „Chunk-Bezeichner" mit drei Bit, der den zurückgegebenen Chunk bezeichnet; und
3. Einen „Thread-Bezeichner", der den Thread identifiziert, zu dem die zurückgegebenen Daten gehören.

[0048] Anforderungen für den nächsten Befehl werden von dem Zeiger für den nächsten Befehl **100** an den Befehls-TLB **102** ausgebreitet bzw. verbreitet, der eine Adress-Lookup-Operation ausführt und eine physikalische Adresse an den Unified Cache **44** liefert. Der Unified Cache **44** liefert einen entsprechenden Makrobefehl an den Befehls-Streaming-Puffer **106**. Jede Anforderung des nächsten Befehls wird ebenfalls direkt von dem Zeiger auf den nächsten Befehl **100** zu dem Befehls-Streaming-Puffer **106** ausgebreitet, so dass der Befehls-Streaming-Puffer **106** den Thread identifizieren kann, dem ein Makrobefehl angehört, der von dem Unified Cache **44** empfangen worden ist. Danach werden die Makrobefehle sowohl von den ersten als auch den zweiten Threads von dem Befehls-Streaming-Puffer **106** an den Befehlsvordecodierer **108** ausgegeben, der eine Reihe von Längenberechnungs- und Bytemarkierungsoperationen in Bezug auf einen empfangenen Befehlsstrom (von Makrobefehlen) ausführt. Im Besonderen erzeugt der Befehlsvordecodierer **108** eine Reihe von Bytemarkierungsvektoren, die unter anderem der Abgrenzung von Makrobefehlen in dem Befehlsstrom dienen, der an die Befehlsleitlogik **110** ausgebreitet wird.

[0049] Die Befehlsleitlogik **1210** verwendet danach die Bytemarkierungsvektoren zum Leiten diskreter Makrobefehle zu dem Befehlsdecodierer **112** zum Zwecke der Decodierung. Makrobefehle werden ferner von der Befehlsleitlogik **110** zu der Verzweigungsadressberechnungseinheit **114** ausgebreitet, und zwar zu dem Zweck der Berechnung von Verzweigungsadressen. Mikrobefehle werden danach von dem Befehlsdecodierer **112** der Trace-Delivery-Engine **60** zugestellt.

[0050] Während der Decodierung werden Flussmarkierungen jedem Mikrobefehl zugeordnet, in den ein Makrobefehl übersetzt wird. Eine Flussmarkierung zeigt eine Eigenschaft des zugeordneten Mikrobefehls an und kann zum Beispiel den zugeordneten Mikrobefehl als den ersten oder letzten Mikrobefehl in einer Mikrocodesequenz anzeigen, die einen Makrobefehl darstellt. Die Flussmarkierungen weisen eine Flussmarkierung „Anfang von Makrobefehl" (BOM) und „Ende von Makrobefehl" (EOM) auf. Gemäß der vorliegenden Erfindung kann der Decodierer **112** ferner die Mikrobefehle so decodieren, dass sie Flussmarkierungen mit gemeinsam genutzten Ressourcen (Multiprozessor) und Synchronisierungs-Flussmarkierungen (SYNC) aufweisen, die ihnen zugeordnet sind. Im Besonderen identifiziert eine Flussmarkie-

rung für eine gemeinsam genutzte Ressource einen Mikrobefehl als eine Position in einem bestimmten Thread, an der der Thread mit weniger negativen Konsequenzen als an anderer Stelle in dem Thread unterbrochen (z. B. neu gestartet oder angehalten) werden kann. Der Decodierer **112** ist in einem exemplarischen Ausführungsbeispiel der vorliegenden Erfindung so gestaltet, dass er Mikrobefehle markiert bzw. kennzeichnet, die das Ende oder den Anfang eines übergeordneten Makrobefehls umfassen, mit einer Flussmarkierung für eine gemeinsam genutzte Ressource sowie unterbrochene Punkte in längeren Mikrocodesequenzen. Eine Synchronisierungs-Flussmarkierung identifiziert einen Mikrobefehl als eine Position in einem bestimmten Thread, an der der Thread mit einem anderen Thread synchronisiert werden kann, zum Beispiel als Reaktion auf einen Synchronisierungsbefehl in dem anderen Thread. Für die Zwecke der vorliegenden Patentschrift bezieht sich der Begriff „synchronisieren" auf die Identifikation zumindest eines ersten Punktes in mindestens einem Thread, an dem der Prozessorzustand modifiziert werden kann in Bezug auf diesen Thread und/oder zumindest einen weiteren Thread mit reduzierter oder geringerer Unterbrechung des Prozessors im Verhältnis zu einem zweiten Punkt in dem Thread oder in einem anderen Thread.

[0051] Der Decodierer **112** ist in einem exemplarischen Ausführungsbeispiel der vorliegenden Erfindung so konstruiert, dass er Mikrobefehle kennzeichnet, die sich an ausgewählten Makrobefehlsgrenzen befinden, an denen zwischen den Threads gemeinsam genutzte Zustände, die gemeinsam in dem gleichen Prozessor existieren, durch einen Thread verändert werden können, ohne die Ausführung anderer Threads nachteilig zu beeinflussen.

[0052] Von der Mikrobefehls-Translation-Engine **54** werden decodierte Befehle (d. h. Mikrobefehle) zu der Trace-Delivery-Engine **60** gesendet. Die Trace-Delivery-Engine **60** weist einen Trace-Cache **62**, eine Trace-Verzweigungsvorhersageeinrichtung (BTB) **64**, einen Mikrocode-Sequencer **66** und eine Mikrocodewarteschlange (uop-Warteschlange) **68** auf. Die Trace-Delivery-Engine **60** fungiert als ein Mikrobefehls-Cache und ist die primäre Quelle von Mikrobefehlen für eine nachgeschaltete Ausführungseinheit **70**. Durch die Bereitstellung einer Mikrobefehls-Caching-Funktion in der Prozessor-Pipeline ermöglicht die Trace-Delivery-Engine **60** und im Besonderen der Trace-Cache **62** es, dass von der Mikrobefehls-Translation-Engine **54** vorgenommene Übersetzungsarbeiten optimiert bzw. skaliert werden können, um die Mikrobefehlsbandbreite zu erhöhen. In dem exemplarischen Ausführungsbeispiel kann der Trace-Cache **62** einen 256 set, 8 way set assoziierten Speicher umfassen. Der Begriff „Trace" kann sich in dem vorliegenden exemplarischen Ausführungsbeispiel auf eine Sequenz von Mikrobefehlen beziehen, die in Einträgen des Trace-Cache **62** gespeichert sind, wobei jeder Eintrag Zeiger auf vorangehende und fortschreitende Mikrobefehle aufweist, welche die Spur bzw. Trace umfassen. Auf diese Weise erleichtert der Trace-Cache **62** die Hochleistungsablaufsteuerung dahingehend, dass die Adresse des Eintrags, auf den als nächstes zugegriffen wird, für die Zwecke der Ermittlung eines folgenden Mikrobefehls bekannt ist, bevor ein aktueller Zugriff abgeschlossen ist. Traces bzw. Spuren können in einem Ausführungsbeispiel als „Blöcke" von Befehlen betrachtet werden, die sich durch Trace-Heads voneinander unterscheiden und enden, wenn auf eine andere indirekte Verzweigung getroffen wird oder wenn einer der zahlreichen vorhandenen Schwellenwertbedingungen erreicht wird, wie etwa die Anzahl der konditionierten Verzweigungen, die in einer einzelnen Spur vorgesehen sein können, oder die maximale Anzahl der Mikrobefehle insgesamt, die eine Spur umfassen kann.

[0053] Die Trace-Cache-Verzweigungsvorhersageeinrichtung **64** stellt lokale Verzweigungsvorhersagen in Bezug auf Spuren in dem Trace-Cache **62** bereit. Der Trace-Cache **62** und der Mikrocode-Sequencer **66** stellen Mikrobefehle an die Mikrocodewarteschlange **68** bereit, von wo die Mikrobefehle danach einem Out-of-Order-Ausführungs-Cluster zugeführt werden. Gemäß der Abbildung weist der Mikrocode-Sequencer **66** ferner eine Reihe von Ereignisbehandlungsroutinen **67** auf, ausgeführt in Mikrocode, welche eine Reihe von Operationen in dem Prozessor **30** ausführen, und zwar als Reaktion auf das Eintreten eines Ereignisses wie etwa einer Ausnahme oder einer Unterbrechung. Die Ereignisbehandlungsroutinen **67** werden, wie dies nachstehend im Text näher beschrieben wird, durch einen Ereignisdetektor **188** aufgerufen, der in einer Registerumbenennungseinrichtung **74** in dem Back-End des Prozessors **30** enthalten ist.

[0054] Der Prozessor **30** kann so betrachtet werden, dass er ein In-Order-Front-End aufweist, das die Busschnittstellenseinheit **32**, die Speicherausführungseinheit **42**, die Mikrobefehls-Translation-Engine **54** und die Trace-Delivery-Engine **60** ebenso umfasst wie ein Out-of-Order-Back-End, das nachstehend im Text näher beschrieben wird.

[0055] Aus der Mikrocodewarteschlange **68** ausgegebene Mikrobefehle werden in einem Out-of-Order-Cluster **71** empfangen, der einen Scheduler **72**, eine Registerumbenennungseinrichtung **74**, eine Zuordnungseinrichtung **76**, einen Neuordnungspuffer **78** und eine Wiedergabewarteschlange **80** umfasst. Der Scheduler **72** weist eine Reihe von Reservierungsstationen auf und arbeitet so, dass er Mikrobefehle für die Ausführung durch die Ausführungseinheit **70** hinsichtlich der Laufzeit plant bzw. einteilt und ausgibt. Die Registerumbenennungseinrichtung **74**

führt eine Registerumbenennungsfunktion in Bezug auf verdeckte ganzzahlige und Gleitkommaregister aus (die an Stelle jedes der acht universellen Register oder jedes der acht Gleitkommaregister eingesetzt werden kann, wenn ein Prozessor **30** einen Befehlssatz mit Intel-Architektur ausführt). Die Zuordnungseinrichtung **76** arbeitet so, dass Ressourcen der Ausführungseinheit **70** und des Clusters **71** Mikrobefehlen gemäß der Verfügbarkeit und den Anforderungen zugeordnet werden. Für den Fall, dass nicht ausreichend Ressourcen für die Verarbeitung eines Mikrobefehls zur Verfügung stehen, ist die Zuordnungseinrichtung **76** verantwortlich für die Aktivierung eines Stopp- bzw. Anhaltesignals **82**, das durch die Trace-Delivery-Engine **60** an die Mikrobefehls-Translation-Engine **54** ausgebreitet wird, wie dies unter **58** dargestellt ist. Mikrobefehle, deren Quellenfelder durch die Registerumbenennungseinrichtung **74** angepasst worden sind, werden in strikter Programmreihenfolge in einem Neuordnungspuffer **78** platziert. Wenn Mikrobefehle in dem Neuordnungspuffer **78** die Ausführung abgeschlossen haben und für einen Rückzug bereit sind, werden sie aus einem Neuordnungspuffer entfernt und "In-Order" bzw. in der entsprechenden Reihenfolge abgerufen (d. h. gemäß einer ursprünglichen Programmreihenfolge). Die Wiedergabewarteschlange **80** breitet Mikrobefehle aus, die von der Ausführungseinheit **70** wiedergegeben werden sollen.

[0056] Die Ausführungseinheit **70** weist in der Abbildung eine Gleitkomma-Ausführungs-Engine **84**, eine ganzzahlige Ausführungs-Engine **86** und einen Level-0-Daten-Cache **88** auf. In einem exemplarischen Ausführungsbeispiel, in dem der Prozessor **30** den Intel-Architektur-Befehlssatz ausführt, kann die Gleitkomma-Ausführungs-Engine **84** ferner auch MMX®-Befehle und Streaming SIMD (Single Instruction, Multiple Data) Erweiterungen (SSEs) ausführen.

Multithreading-Implementierung

[0057] In dem exemplarischen Ausführungsbeispiel des Prozessors **30** aus der Abbildung aus Fig. 2 kann eine begrenzte Duplizierung oder Replizierung von Ressourcen existieren, um die Multithreading-Fähigkeit zu unterstützen, und somit ist es erforderlich, zwischen den Threads ein gewisses Maß der gemeinsamen Nutzung von Ressourcen zu implementieren. Hiermit wird festgestellt, dass die eingesetzte Methode zur gemeinsamen Nutzung von Ressourcen abhängig ist von der Anzahl von Threads, die der Prozessor gleichzeitig verarbeiten kann. Da funktionale Einheiten in einem Prozessor für gewöhnlich eine gewisse Pufferfunktionalität (oder Speicherfunktionalität) und Ausbreitungsfunktionalität bereitstellen, kann der Aspekt der gemeinsamen Nutzung von Ressourcen so betrachtet werden, dass er folgendes umfasst: (1) das Speichern und (2) das Verarbeiten/Ausbreiten von Komponenten, welche Bandbreite gemeinsam nutzen. In einem Prozessor, der die gleichzeitige Verarbeitung von zwei Threads unterstützt, können die Pufferresourcen in verschiedenen funktionalen Einheiten statisch oder logisch zwischen zwei Threads aufgeteilt werden. In ähnlicher Weise muss die durch einen Pfad für die Ausbreitung von Informationen zwischen zwei funktionalen Einheiten bereitgestellte Bandbreite zwischen den beiden Threads aufgeteilt und zugeordnet werden. Da diese Aspekte zur gemeinsamen Nutzung von Ressourcen an einer Reihe von Positionen in einer Prozessor-Pipeline auftreten können, können unterschiedliche Methoden zur gemeinsamen Nutzung von Ressourcen an diesen verschiedenen Stellen bzw. Positionen gemäß den Vorgaben und den Eigenschaften der jeweiligen Position eingesetzt werden. Hiermit wird festgestellt, dass sich unterschiedliche Systeme zur gemeinsamen Nutzung von Resourcen für verschiedene Positionen eignen können in Anbetracht der verschiedenen Funktionalitäten und Betriebseigenschaften.

[0058] Die Abbildung aus Fig. 3 zeigt ein Blockdiagramm, das ausgewählte Komponenten für ein Ausführungsbeispiel des Prozessors **30** gemäß der Veranschaulichung aus Fig. 2 veranschaulicht und verschiedene funktionale Einheiten darstellt, die eine Pufferkapazität bereitstellen, die logisch aufgeteilt ist, so dass zwei Threads (d. h. Thread 0 und Thread 1) berücksichtigt werde. Die logische Aufteilung der beiden Threads der Puffer-(oder Speicher-) und Verarbeitungsfunktionen einer funktionalen Einheit können erreicht werden, indem eine erste vorbestimmte Reihe von Einträgen in einer Pufferressource einem ersten Thread zugeordnet wird, und wobei eine zweite vorbestimmte Reihe von Einträgen in der Pufferressource einem zweiten Thread zugeordnet wird. In alternativen Ausführungsbeispielen kann das Puffer ebenfalls dynamisch gemeinsam genutzt werden. Im Besonderen kann dies erreicht werden durch das Bereitstellen von zwei Lese- und Schreib-Zeigern, einem ersten Paar von Lese- und Schreib-Zeigern, die einem ersten Thread zugeordnet sind, und einem zweiten Paar von Lese- oder Schreib-Zeigern, die einem zweiten Thread zugeordnet sind. Die erste Reihe von Lese- und Schreib-Zeigern kann auf eine erste vorbestimmte Anzahl von Einträgen in einer Pufferressource beschränkt sein, während die zweite Reihe von Lese- und Schreib-Zeigern auf eine zweite vorbestimmte Anzahl von Einträgen in der gleichen Pufferressource beschränkt sein kann. In dem veranschaulichten Ausführungsbeispiel sind der Befehls-Streaming-Puffer **106**, der Trace-Cache **62** und eine Befehlswarteschlange **103** so dargestellt, dass sie jeweils eine Speicherkapazität bereitstellen, die logisch zwischen den ersten und zweiten Threads aufgeteilt ist.

Der Out-of-Order-Cluster (**71**)

[0059] Die Abbildung aus Fig. 4 zeigt ein Blockdiagramm, das weitere Einzelheiten eines Ausführungsbeispiels des Out-of-Order-Clusters **71** veranschaulicht. Der Cluster **71** stellt die Funktionalität der Reservierungsstation, der Registerumbenennung, der Wiedergabe und des Rückzugs in dem Prozessor **30** bereit. Der Cluster **71** empfängt Mikrobefehle von der Trace-Delivery-Engine **60**, weist diesen Mikrobefehlen Ressourcen zu, benennt Quellen- und Zielregister für jeden Mikrobefehl um, teilt Mikrobefehle für die Ausgabe an die entsprechenden Ausführungseinheiten **70** ein, behandelt Mikrobefehle, die aufgrund Datenspekulation wiedergegeben werden, und führt Mikrobefehle schließlich zurück (d. h. weist den Mikrobefehlen einen dauerhaften Architekturzustand zu).

[0060] An dem Cluster **71** empfangene Mikrobefehle werden gleichzeitig einer Register-Alias-Tabelle **120** und einer Zuordnungs- und freien Listenverwaltungslogik **122** zugeführt. Die Register-Alias-Tabelle **120** ist zuständig für die Übersetzung logischer Registernamen in physikalische Registeradressen, welche von dem Scheduler **72** und den Ausführungseinheiten **70** verwendet werden. Im Besonderen benennt in Bezug auf die Abbildung aus Fig. 5 die Register-Alias-Tabelle **120** ganzzahlige, Gleitkomma- und Segmentregister in einer physikalischen Registerdatei **124** um. Die Registerdatei **124** weist in der Abbildung **126** physikalische Register auf, die ein Aliasing an acht (8) Architekturregister aufweisen. In dem veranschaulichten Ausführungsbeispiel weist die Register-Alias-Tabelle **120** sowohl eine Front-End-Tabelle **126** als auch eine Back-End-Tabelle **128** zur Nutzung durch die entsprechenden Front-Ends und Back-Ends des Prozessors **30** auf. Jeder Eintrag in der Register-Alias-Tabelle **120** ist einem Architekturregister zugeordnet oder wird als ein solches gesehen und weist einen Zeiger **130** auf, der auf eine Position in der Registerdatei **124** zeigt, an der die dem relevanten Architekturregister zugeordneten Daten gespeichert werden. Auf diese Weise können die Herausforderungen adressiert werden, die durch ältere Mikroprozessorarchitektur vorgesehen werden, welche eine verhältnismäßig kleine Anzahl von Architekturregistern spezifiziert.

[0061] Die Zuordnungs- und freie Listenverwaltungslogik **122** ist zuständig für die Ressourcenzuordnung und die Zustandswiederherstellung in dem Cluster **71**. Die Logik **122** weist die folgenden Ressourcen jedem Mikrobefehl zu:

1. Eine Sequenznummer, die für jeden Mikrobefehl erteilt wird, um deren logische Reihenfolge in einem Thread zu verfolgen, wenn der Mikrobefehl in dem Cluster **71** verarbeitet wird. Die jedem Mikrobefehl zugeordnete Sequenznummer wird gemeinsam mit Statusinformationen für den Mikrobefehl in einer Tabelle (in Fig. 10 unten dargestellt) in dem Neuordnungspuffer **162** gespeichert.
2. Einen Eintrag in der freien Listenverwaltung, der für jeden Mikrobefehl vergeben wird, um eine Verfolgung der Historie des Mikrobefehls und deren Wiederherstellung für den Fall einer Zustandswiederherstellungsoperation zu ermöglichen.
3. Einen Eintrag in dem Neuordnungspuffer (ROB), wobei der Eintrag durch die Sequenznummer indexiert ist.
4. Einen Eintrag in der physikalischen Registerdatei **124** (bekannt als „Marble"), in der der Mikrobefehl nützliche Ergebnisse speichern kann.
5. Einen Lastpuffereintrag (nicht abgebildet).
6. Einen Anhaltepuffereintrag (nicht abgebildet).
7. Einen Befehlswarteschlangeneintrag (z. Beispiel in einer Speicherbefehlswarteschlange oder einer universellen Befehlsadresswarteschlange, wie dies nachstehend im Text näher beschrieben wird).

[0062] Für den Fall, dass die Logik **122** nicht in der Lage ist, die erforderlichen Ressourcen für eine empfangene Sequenz von Mikrobefehlen zu erhalten, fordert die Logik **122** an, dass die Trace-Delivery-Engine **60** die Lieferung von Mikrobefehlen anhält, bis ausreichende Ressourcen verfügbar werden. Diese Anforderung wird kommuniziert, indem das Anhaltesignal **82** aktiviert ist, wie dies in Fig. 2 dargestellt ist.

[0063] In Bezug auf die Zuordnung eines Eintrags in der Registerdatei **124** zu einem Mikrobefehl zeigt die Abbildung aus Fig. 5 eine Trash-Heap-Aray **132**, welche einen Datensatz von Einträgen in der Registerdatei **124** verwaltet, die nicht Architekturregistern zugeordnet sind (d. h. für die es keine Zeiger in der Register-Alias-Tabelle **120** gibt). Die Logik **122** greift auf die Trash-Heap-Array **132** zu, um Einträge in der Registerdatei **124** zu identifizieren, die zur Verfügung stehen für die Zuordnung zu einem empfangenen Mikrobefehl. Die Logik **122** ist auch zuständig für die erneute Beanspruchung von Einträgen in der Registerdatei **124**, die verfügbar werden.

[0064] Die Logik **122** verwaltet ferner einen freien Listenmanager (FLM) **134**, um die Nachverfolgung von Architekturregistern zu ermöglichen. Im Besonderen verwaltet der freie Listenmanager **134** eine Historie der Veränderungen der Register-Alias-Tabelle **120**, wenn dieser Mikrobefehle zugeordnet werden. Der freie Listenmanager **134** stellt die Fähigkeit bereit, die Register-Alias-Tabelle **120** „abzuwickeln", um auf einen nicht spekulativen Zustand zu zeigen, wenn eine falsche Vorhersage oder ein Ereignis gegeben ist. Der freie Listenmanager **134** „altert" auch die Speicherung von Daten in den Einträgen der Registerdatei **124**, um zu garantieren, dass alle Zustandsinformationen aktuell sind. Schließlich werden bei einem Rückzug physikalische Registerbezeichner von dem freien Listenmanager **134** zu der

Trash-Heap-Array **132** übertragen, um sie einem weiteren Mikrobefehl zuzuordnen.

[0065] Eine Befehlswarteschlangeneinheit **136** liefert Mikrobefehle an eine Scheduler und Scoreboard Unit (SSU) **138** in sequentieller Programmreihenfolge, und sie hält und gibt Mikrobefehlsinformationen aus, die von den Ausführungseinheiten **70** benötigt werden. Die Befehlswarteschlangeneinheit **136** kann zwei unterschiedliche Strukturen aufweisen, nämlich eine Befehlswarteschlange (IQ) **140** und eine Befehlsadresswarteschlange (IAQ) **142**. Die Befehlsadresswarteschlangen **142** sind kleine Strukturen, die so gestaltet sind, dass sie kritische Informationen (z. B. Mikrobefehlsquellen, Ziele bzw. Destinationen und Latenz) nach Bedarf der Einheit **138** zuführen. Die Befehlsadresswarteschlange **142** kann ferner eine Speicherbefehlsadresswarteschlange (MIAQ) umfassen, die Informationen für Speicheroperationen in die Warteschlange einstellt, und eine universelle Befehlsadresswarteschlange (GIAQ), welche Informationen für nicht-speicherbezogene Operationen in die Warteschlange einstellt. Die Befehlswarteschlange **140** speichert weniger kritische Informationen, wie etwa Operationscode und direkte Daten für Mikrobefehle. Für Mikrobefehle werden Zuordnungen von der Befehlswarteschlangeneinheit **136** aufgehoben, wenn die relevanten Mikrobefehle gelesen und in die Scheduler und Scoreboard Unit **138** geschrieben werden.

[0066] Die Scheduler und Scoreboard Unit **138** ist zuständig für das Scheduling von Mikrobefehlen für die Ausführung, indem die Zeit bestimmt wird, zu der jede der Mikrobefehlsquellen bereit sein kann und zu der die entsprechende Ausführungseinheit für die Ausgabe zur Verfügung steht. Die Einheit **138** umfasst gemäß der Abbildung aus **Fig. 4** ein Registerdatei-Scoreboard **144**, einen Speicher-Scheduler **146**, einen Matrix-Scheduler **148**, einen langsamen Befehls Scheduler **150** und einen Gleitkomma-Scheduler **152**.

[0067] Die Einheit **138** bestimmt, wann das Quellenregister bereit ist, indem die in dem Registerdatei-Scoreboard **144** verwalteten Informationen geprüft werden. Zu diesem Zweck weist das Registerdatei-Scoreboard **144** in einem Ausführungsbeispiel 256 Bits auf, welche die Verfügbarkeit von Datenressourcen verfolgen, entsprechend jedem Register in der Registerdatei **124**. Zum Beispiel können die Scoreboard-Bits für einen bestimmten Eintrag in der Registerdatei **124** nach der Zuordnung der Daten zu dem relevanten Eintrag oder einer Schreiboperation in die Einheit **138** gelöscht werden.

[0068] Der Speicher-Scheduler **146** puffert Speicherklassen-Mikrobefehle, prüft die Ressourcenverfügbarkeit und führt dann ein Scheduling der Speicherklassen-Mikrobefehle durch. Der Matrix-Scheduler **148** umfasst zwei eng begrenzte ALU-Scheduler (ALU als englische Abkürzung von Arithmetic Logic Unit bzw. arithmetische Logikeinheit), die ein Scheduling von abhängigen Mikrobefehlen antiparallelgeschaltet ermöglichen. Der Gleitkomma-Scheduler **152** puffert und teilt Gleitkomma-Mikrobefehle ein, während der langsame Befehls-Scheduler **150** Mikrobefehle einteilt, die nicht von den oben genannten Scheduler behandelt werden.

[0069] Eine Prüf-, Wiedergabe- und Rückzugs-Einheit (CRU) **160** weist gemäß der Abbildung einen Neuordnungspuffer **162**, eine Prüfeinrichtung **164**, eine Staging-Warteschlange **166** und eine Rückzugsregelschaltung **168** auf. Die Einheit **160** weist drei Hauptfunktionen auf, nämlich eine Prüffunktion, eine Wiedergabefunktion und eine Rückzugsfunktion. Im Besonderen umfassen die Prüf- und Wiedergabefunktionen die erneute Ausführung von Mikrobefehlen, die nicht korrekt ausgeführt worden sind. Die Rückzugsfunktion umfasst die Zuordnung eines architektonischen In-Order-Zustands an den Prozessor **30**. Im Besonderen arbeitet die Prüfeinrichtung **164** so, dass garantiert wird, dass der Mikrobefehl die richtigen Daten ordnungsgemäß ausgeführt hat. Für den Fall, dass der Mikrobefehls nicht mit den korrekten Daten ausgeführt worden ist (z. B. aufgrund einer falsch vorhergesagten Verzweigung), wird der relevante Mikrobefehl wiedergegeben, um eine Ausführung mit den richtigen Daten zu gewährleisten.

[0070] Der Neuordnungspuffer **162** ist zuständig für die Zuordnung bzw. Zuweisung des architektonischen Zustands an den Prozessor **30**, indem Mikrobefehle in Programmreihenfolge zurückgezogen werden. Ein Rückzugszeiger **182**, der durch eine Rückzugssteuerschaltung **168** erzeugt wird, zeigt einen Eintrag in dem Neuordnungspuffer **162** an, der zurückgezogen wird. Wenn sich der Rückzugszeiger **182** in einem Eintrag an dem Mikrobefehl vorbei bewegt, wird danach der entsprechende Eintrag in dem freien Listenmanager **134** freigegeben, und der relevante Registerdateieintrag kann jetzt neu angefordert und zu der Trash-Heap-Array **132** übertragen werden. Die Rückzugssteuerschaltung **168** implementiert gemäß der Abbildung eine aktive Thread-Zustandsmaschine **171**, deren Zweck und Funktionsweise nachstehend beschrieben wird. Die Rückzugssteuerschaltung **168** steuert die Zuweisung spekulativer Ergebnisse, die in dem Neuordnungspuffer **162** gehalten werden, an den architektonischen Zustand in der Registerdatei **124**.

[0071] Der Neuordnungspuffer **162** ist ferner zuständig für die Behandlung interner und externer Ereignisse, wie dies nachstehend im Text näher beschrieben wird. Nach dem Erkennen des Auftretens eines Ereignisses durch den Neuordnungspuffer **162** wird ein Signal „Nuke" **170** aktiviert. Das Nuke-Signal **170** weist den Effekt auf, dass alle Mikrobefehle aus

der Prozessor-Pipeline gelöscht werden, die sich gerade in der Bewegung befinden. Der Neuordnungspuffer **162** stellt ferner für die Trace-Delivery-Engine **60** eine Adresse bereit, von der mit der Ablaufsteuerung der Mikrobefehle zur Bedienung bzw. Behandlung des Ereignisses begonnen wird (d. h. von der eine in Mikrocode ausgeführte Ereignisbehandlungsroutine **67** ausgegeben wird).

Der Neuordnungspuffer (**162**)

[0072] Die Abbildung aus Fig. 6A zeigt ein Blockdiagramm, das weitere Einzelheiten in Bezug auf ein exemplarisches Ausführungsbeispiel des Neuordnungspuffers **162** veranschaulicht, der logisch aufgeteilt ist, um mehrere Threads in dem multithreading-fähigen Prozessor zu bedienen bzw. zu behandeln. Im Besonderen weist der Neuordnungspuffer **162** eine Neuordnungstabelle **180** auf, die logisch so partitioniert werden kann, dass Einträge für die ersten und zweiten Threads berücksichtigt werden, wenn der Prozessor **30** in einem multithreading-fähigen Modus arbeitet. Bei einem Betrieb in einem Modus mit nur einem einzigen Thread kann die ganze Tabelle **180** eingesetzt werden, um den einzelnen Thread zu bedienen. Die Tabelle **180** umfasst in einem Ausführungsbeispiel eine unitäre Speicherstruktur, die bei einem Betrieb in einem multithreading-fähigen Modus referenziert wird durch zwei (2) Rückzugszeiger **182** und **183**, die auf vorbestimmte und getrennte Gruppen von Einträgen in der Tabelle **180** begrenzt sind. Bei einem Betrieb in einem Modus mit nur einem Thread wird in ähnlicher Weise durch einen einzigen Rückzugszeiger **182** auf die Tabelle Bezug genommen. Die Tabelle **180** weist einen Eintrag auf, der jedem Eintrag der Registerdatei **124** entspricht, und speichert eine Sequenznummer und Statusinformationen in Form von Fehlerinformationen, eine logische Zieladresse und ein gültiges Bit für jeden Mikrobefehlsdateneintrag in der Registerdatei **124**. Die Einträge in der Tabelle **180** werden jeweils indexiert durch die Sequenznummer, die einen eindeutigen Bezeichner für jeden Mikrobefehl darstellt. Einträge in der Tabelle **180** werden gemäß den Sequenznummern zugeordnet und deren Zuordnungen aufgehoben in einer sequentiellen und In-Order-Vorgehensweise. Zusätzlich zu anderen Flussmarkierungen ist die Tabelle **180** ferner so dargestellt, dass sie eine Flussmarkierung **184** für gemeinsam genutzte Ressourcen speichert und eine Synchronisierungs-Flussmarkierung **186** für jeden Mikrobefehl.

[0073] Der Neuordnungspuffer **162** weist einen Ereignisdetektor **188** auf, der so gekoppelt ist, dass er Unterbrechungsanforderungen in Form von Unterbrechungsvektoren empfängt und ferner auf Einträge in der Tabelle **180** zugreift, referenziert durch die Rückzugszeiger **182** und **183**. Der Ereignisdetektor **188** gibt gemäß der Abbildung ferner ein Nuke-Signal **170** und ein Löschsignal **172** aus.

[0074] In der Annahme, dass ein bestimmter Mikrobefehl für einen spezifischen Thread (z. B. Thread 0) keine falsche Verzweigungsvorhersage, Ausnahme oder Unterbrechung erfährt, so werden die in dem Eintrag in der Tabelle **180** für den spezifischen Befehl gespeicherten Informationen in den architektonischen Zustand zurückgestellt, wenn der Rückzugszeiger **182** oder **183** heraufgesetzt wird, um den relevanten Eintrag zu adressieren. In diesem Fall setzt eine Befehlszeigerberechnungseinrichtung **190**, die einen Bestandteil der Rückzugssteuerschaltung **168** bildet, den Makro- oder Mikrobefehlszeiger herauf, um auf folgendes zu zeigen: (1) eine Verzweigungszieladresse, die in dem entsprechenden Eintrag in der Registerdatei **124** spezifiziert ist, oder (2) den nächsten Makro- bzw. Mikrobefehl, wenn eine Verzweigung nicht vorgenommen wird.

[0075] Wenn eine falsche Verzweigungsvorhersage eingetreten ist, werden die Informationen über das Fehlerinformationsfeld zu der Rückzugssteuerschaltung **168** und dem Ereignisdetektor **188** übertragen. In Anbetracht der durch die Fehlerinformationen angezeigten falschen Verzweigungsvorhersage kann der Prozessor **30** mindestens einige falsche Befehle erfasst haben, die die Prozessor-Pipeline durchdrungen haben. Da Einträge in der Tabelle **180** in sequentieller Reihenfolge zugeordnet werden, stellen alle Einträge nach dem falsch vorhergesagten Verzweigungsmikrobefehl Mikrobefehle dar, die durch den falsch vorhergesagten Verzweigungsbefehlsfluss verdorben worden sind. Als Reaktion auf den versuchten Rückzug eines Mikrobefehls, für den eine falsch vorhergesehene Verzweigung in den Fehlerinformationen registriert ist, aktiviert der Ereignisdetektor **188** das Löschsignal **172**, das das vollständige Out-of-Order-Back-End des Prozessors aller Zustände löscht und somit das Out-of-Order-Back-End aller Zustände leert, welche von Befehlen resultieren, die einem falsch vorhergesehen Mikrobefehl folgen. Die Aktivierung des Lösch- bzw. Auslösesignals **172** blockiert zudem die Ausgabe nacheinander erfasster Mikrobefehle, die sich in dem In-Order-Front-End des Prozessors **30** befinden können.

[0076] In der Rückzugssteuerschaltung **168** stellt nach der Benachrichtigung über eine falsch vorhergesehene Verzweigung durch die Fehlerinformationen eines rückgezogenen Mikrobefehls die IP-Berechnungseinrichtung **190** sicher, dass die Befehlszeiger **179** und/oder **181** aktualisiert werden, um den richtigen Befehlszeigerwert darzustellen. Abhängig davon, ob die Verzweigung vorgenommen oder nicht vorgenommen werden soll, aktualisiert die IP-Berechnungseinrichtung **190** die Befehlszeiger **179** und/oder **181** mit den Ergebnisdaten aus dem Registerdateieintrag gemäß dem relevanten Eintrag der Tabelle **180**, oder inkrementiert die Befehlszeiger **179** und **181**, wenn der Verzweigung nicht gefolgt worden ist.

[0077] Der Ereignisdetektor **188** weist ferner eine Reihe von Registern **200** zur Verwaltung von Informationen zu Ereignissen auf, die für jeden von mehreren Threads detektiert worden sind. Die Register **200** weisen ein Ereignisinformationsregister **202**, ein Register **204** für anstehende Ereignisse, ein Ereignissperrregister **206** und ein Unwind- bzw. Abwickelungsregister **208** und ein Pin-Zustandsregister **210** auf. Jedes der Register **202**–**210** ist in der Lage, Informationen zu speichern, die ein Ereignis betreffen, das für einen spezifischen Thread erzeugt worden ist. Demgemäß können Ereignisinformationen für mehrere Threads von den Registern **200** verwaltet werden.

[0078] Die Abbildung aus Fig. 6B zeigt eine schematische Darstellung eines exemplarischen Registers **204** für ein anstehendes Ereignis und eines exemplarischen Ereignissperrregisters **206** für einen ersten Thread (z. B. T0).

[0079] Die Register **204** und **206** für ein anstehendes Ereignis und zum Sperren von Ereignissen sind vorgesehen für jeden unterstützten Thread in dem multithreading-fähigen Prozessor **30**. Für jeden Thread können eigene Register **204** und **206** bereitgestellt werden, oder alternativ kann ein einzelnes physikalisches Register logisch aufgeteilt werden, um mehrere Threads zu unterstützen.

[0080] Das exemplarische Register **204** für anstehende bzw. ausstehende Register enthält ein Bit oder ein anderes Datenelement für jeden Ereignistyp, der von dem Ereignisdetektor **188** detektiert wird (z. B. die nachstehend in Bezug auf die Abbildung aus Fig. 8 beschriebenen Ereignisse). Diese Ereignisse können interne Ereignisse darstellen, die intern in dem Prozessor **30** erzeugt werden, oder es kann sich um externe Ereignisse handeln, die außerhalb des Prozessors **30** erzeugt werden (z. B. Pin-Ereignisse, die von dem Prozessorbus empfangen werden). Das Register **204** für anstehende Ereignisse für jeden Thread weist in dem veranschaulichten Ausführungsbeispiel kein Bit für ein Writeback-Ereignis (Zurückschreiben) auf, da derartige Ereignisse nicht Thread-spezifisch sind und somit nicht in dem Register für anstehende Ereignisse in die „Warteschlange" eingestellt werden. Zu diesem Zweck kann der Ereignisdetektor **188** eine Writeback-Detektierungslogik **205** aufweisen, die ein Writeback-Signal aktiviert, wenn ein Writeback-Ereignis detektiert wird. Die Bits in dem Register **204** für ein anstehendes Ereignis für jeden Thread werden durch den Ereignisdetektor **188** gesetzt, der ein Latch auslöst, das das entsprechende Bit in dem Register **204** für anstehende Ereignisse auslöst. In einem exemplarischen Ausführungsbeispiel stellt ein gesetztes Bit, das einem vorbestimmten Ereignis zugeordnet ist, in dem Register **204** für ein anstehendes Ereignis eine Anzeige bereit, dass ein Ereignis des relevanten Typs ansteht, wie dies nachstehend im Text näher beschrieben wird.

[0081] Das Ereignissperrregister **206** für jeden Thread weist in ähnlicher Weise ein Bit oder eine andere Datenstruktur für jeden Ereignistyp auf, der von dem Ereignisdetektor **188** erkannt wird, wobei dieses Bit entweder gesetzt oder zurückgesetzt (d. h. gelöscht) wird, um ein Ereignis als ein Unterbrechungsereignis (Break Event) in Bezug auf den spezifischen Thread aufzuzeichnen. Die entsprechenden Bits in einem Ereignissperrregister **206** werden durch eine Steuerregister-Schreiboperation gesetzt, welche einen speziellen Mikrobefehl nutzt, der einen nicht umbenannten Zustand in dem Prozessor **30** modifiziert. Ein Bit in einem Ereignissperrregister **206** kann in ähnlicher Weise zurückgesetzt (oder gelöscht) werden, unter Verwendung einer Steuerregister-Schreiboperation.

[0082] Ein exemplarischer Prozessor kann ferner bestimmte Modi aufweisen, in denen Bits in dem Ereignissperrregister **206** gesetzt werden können, um ausgewählte Ereignisse in den entsprechenden Modi zu sperren.

[0083] Bits für einen spezifischen Ereignistyp, der in jedem der Register für ein anstehendes Ereignis und ein Ereignissperrregister **204** und **206** für einen spezifischen Thread verwaltet werden, werden an ein UND-Gatter **209** ausgegeben, das wiederum ein Signal **211** für ein detektiertes Ereignis für jeden Ereignistyp ausgibt, wenn die Inhalte der Register **204** und **206** anzeigen, dass der relevante Ereignistyp ansteht und nicht gesperrt wird. Wenn ein Ereignistyp zum Beispiel nicht gesperrt wird, wird bei der Registrierung eines Ereignisses in dem Register **204** für anstehende Ereignisse das Ereignis unverzüglich signalisiert als detektiert durch die Aktivierung des Signals **211** für ein detektiertes Ereignis für den relevanten Ereignistyp. Sollte der Ereignistyp andererseits durch den Inhalt des Ereignissperrregisters **206** gesperrt werden, so wird das Auftreten des Ereignisses in dem Register **204** für anstehende Ereignisse aufgezeichnet, wobei das Signal **211** für ein detektiertes Ereignis nur dann aktiviert wird, wenn das entsprechende Bit in dem Ereignissperrregister **206** gelöscht wird, während das Ereignis in dem Register **204** als anstehend bzw. ausstehend aufgezeichnet wird. Somit kann ein Ereignis in dem Register **204** für anstehende Ereignisse aufgezeichnet werden, wobei das Signal **211** für ein detektiertes Ereignis für das Eintreten eines relevanten Ereignisses nur eine bestimmte Zeit später signalisiert werden kann, wenn die Sperre für das Ereignis für den spezifischen Thread entfernt bzw. aufgehoben wird.

[0084] Die Signale **211** für detektierte Ereignisse für jeden Ereignistyp für jeden Thread werden der Behandlungslogik (Ereignispriorisierungs- und Auswahllogik) und der Taktsteuerlogik zugeführt, wie

dies nachstehend im Text beschrieben wird.

[0085] Eine Ereignisbehandlungsroutine für ein bestimmtes Ereignis ist verantwortlich für das Löschen des entsprechenden Bits in dem Register **204** für ausstehende Ereignisse für einen spezifischen Thread, sobald die Behandlung des Ereignisses abgeschlossen ist. In einem alternativen Ausführungsbeispiel kann das Register für anstehende Ereignisse durch Hardware gelöscht werden.

Eintreten von Ereignissen und Ereignisbehandlung in einer multithreading-fähigen Prozessorumgebung

[0086] Ereignisse in dem multithreading-fähigen Prozessor **30** können von einer Vielzahl von Quellen detektiert und signalisiert werden. Zum Beispiel kann das In-Order-Front-End des Prozessors **30** ein Ereignis anzeigen bzw. signalisieren, und die Ausführungseinheiten **70** können in ähnlicher Weise ein Ereignis signalisieren. Ereignisse können Unterbrechungen und Ausnahmen umfassen. Unterbrechungen bzw. Interrupts sind Ereignisse, die außerhalb des Prozessors **30** erzeugt werden, und sie können von einer Vorrichtung an den Prozessor **30** über einen gemeinsamen Bus (nicht abgebildet) eingeleitet werden. Unterbrechungen können es bewirken, dass der Steuerungsfluss an eine Mikrocode-Ereignisbehandlungsroutine **67** gerichtet wird. Ausnahmen können grob unter anderem in Fehler, Traps und Assists klassifiziert werden. Ausnahmen sind Ereignisse, die für gewöhnlich in dem Prozessor **30** erzeugt werden.

[0087] Ereignisse werden direkt zu dem Ereignisdetektor **188** in dem Neuordnungspuffer **162** kommuniziert, wobei als Reaktion darauf der Ereignisdetektor **188** eine Reihe von Operationen ausführt, die den Thread betreffen, für den oder gegen den der Thread erzeugt worden ist. Auf hoher Ebene unterbricht der Ereignisdetektor **188** als Reaktion auf die Erkennung eines Ereignisses den Rückzug von Mikrobefehlen für den Thread, schreibt die entsprechenden Fehlerinformationen in die Tabelle **180**, aktiviert das Nuke-Signal **170**, ruft eine Ereignisbehandlungsroutine **67** auf, um das Ereignis zu verarbeiten, bestimmt eine Neustartadresse und startet danach die Erfassung von Mikrobefehlen neu. Die Ereignisse können direkt zu dem Ereignisdetektor **188** kommuniziert werden in Form einer Unterbrechungsanforderung (oder eines Unterbrechungssektors) oder durch Fehlerinformationen, die in der Neuordnungstabelle **180** für einen Befehl entweder eines ersten oder eines zweiten Threads aufgezeichnet werden, der zurückgezogen wird.

[0088] Die Aktivierung bzw. Geltendmachung des Nuke-Signals **170** weist den Effekt auf, dass die Zustände des In-Order-Front-Ends und des Out-of-Order-Back-Ends des multithreading-fähigen Prozessors **30** gelöscht werden. Im Besonderen werden für zahlreiche funktionale Einheiten, jedoch nicht unbedingt für alle, der Zustand und die Mikrobefehle als Reaktion auf die Aktivierung des Nuke-Signals **170** gelöscht. Bestimmte Abschnitte des Speicheranordnungspuffers **48** und der Busschnittstelleneinheit **32** werden nicht gelöscht (z. B. zurückgezogene, jedoch nicht erzwungen zurückgeschriebene Speicherungen, Bus-Snoops, etc.). Die Aktivierung des Nuke-Signals **170** hält ferner die Befehlserfassung durch das Front-End an und stoppt die Ablaufsteuerung von Mikrobefehlen in die Mikrocode-Warteschlange **68**. Diese Operation kann ungestört in einem Multiprozessor mit einem Thread oder einem Multiprozessor ausgeführt werden, der den einzelnen Thread ausführt, wobei mehrere Threads existieren und in einem multithreading-fähigen Prozessor **30** verarbeitet werden, wobei die Gegenwart von weiteren Threads nicht ignoriert werden kann, wenn das Eintreten eines Ereignisses in Bezug auf einen einzelnen Thread adressiert wird. Demgemäß schlägt die vorliegende Erfindung ein Verfahren und eine Vorrichtung zur Behandlung eines Ereignisses in einem multithreading-fähigen Prozessor vor, der die Zuständigkeit für die Verarbeitung und die Präsenz mehrerer Threads in dem multithreading-fähigen Prozessor **30** übernimmt, wenn ein Ereignis für einen einzelnen Thread auftritt.

[0089] Die Abbildung aus Fig. 7A zeigt ein Flussdiagramm, das ein Verfahren **220** gemäß einem exemplarischen Ausführungsbeispiel der vorliegenden Erfindung zur Verarbeitung des Auftretens eines Ereignisses in einem multithreading-fähigen Prozessor **30** veranschaulicht. Dass Verfahren **220** beginnt mit dem Block **222** mit dem Detektieren eines ersten Ereignisses für einen ersten Thread durch den Ereignisdetektor **188**. Die Abbildung aus Fig. 8 zeigt eine schematische Darstellung einer Reihe exemplarischer Ereignisse **224**, die durch den Ereignisdetektor **188** in dem Block **222** detektiert werden können. Die in der Abbildung aus Fig. 8 dargestellten Ereignisse sind grob gemäß den Eigenschaften der Reaktionen auf die Ereignisse **224** in Gruppen eingeteilt. Eine erste Gruppe von Ereignissen weist ein Ereignis RESET (Zurücksetzen) **226** und ein Ereignis MACHINE CHECK (Maschinenprüfung) **228** auf, die durch den Ereignisdetektor an mehrere Threads in einem multithreading-fähigen Prozessor **30** auf die nachstehend beschriebene Art und Weise unmittelbar nach der Erkennung signalisiert werden und es bewirken, dass alle Threads gleichzeitig zu der gleichen Ereignisbehandlungsroutine **67** gelangen. Eine zweite Gruppe von Ereignissen beinhaltet ein Ereignis FAULT (Fehler) **230**, ein Ereignis ASSIST (Unterstützung) **232**, ein Ereignis DOUBLE FAULT (doppelter Fehler) **234**, ein Ereignis SHUTDOWN (Herunterfahren) **236** und ein Ereignis SMC (Self Modifying Code bzw. selbstmodifizierender Code) **238**, die jeweils bei dem Rückzug des Mikrobefehls eines bestimmten Threads aufgezeichnet bzw. gemeldet werden, der das Ereignis signalisiert hat. Im Besonderen detektiert der Ereig-

nisdetektor **188** ein Ereignis der zweiten Gruppe, wenn ein Mikrobefehl zurückgezogen wird, für den Fehlerinformationen einen Fehlerzustand anzeigen. Die Erkennung eines Ereignisses durch die zweite Gruppe wird durch den Ereignisdetektor **188** nur dem Thread signalisiert, für den das relevante Ereignis erzeugt worden ist.

[0090] Eine Dritte Gruppe von Ereignissen umfasst ein Ereignis INIT (kurzes Zurücksetzen bzw. Reset) **240**, ein Ereignis INTR (lokale Unterbrechung) **242**, ein Ereignis NMI (nicht maskierbare Unterbrechung) **244**, ein Ereignis DATA BREAKPOINT (Datenunterbrechungspunkt) **246**, ein Ereignis TRACE MESSAGE (Spurnachricht) **248** und ein Ereignis A20M (Adressumlauf) **250**. Die Ereignisse der dritten Gruppe werden beim Rückzug eines Mikrobefehls mit einer Flussmarkierung der Unterbrechungsakzeptanz oder der Trap-Akzeptanz gemeldet. Die Erkennung des Ereignisses der dritten Gruppe wird durch den Ereignisdetektor **188** nur für den Thread signalisiert, für den das relevante Ereignis erzeugt worden ist.

[0091] Eine vierte Gruppe von Ereignissen umfasst ein Ereignis SMI (System Management Interrupt bzw. Systemverwaltungsunterbrechung) **250**, ein Ereignis STOP CLOCK (Takt anhalten) **252** und ein Ereignis PREQ (Probe-Anforderung) **254**. Die Ereignisse der vierten Gruppe werden allen Threads in dem multithreading-fähigen Prozessor **30** signalisiert und gemeldet, wenn einer der mehreren Threads einen Mikrobefehl mit einer entsprechenden Unterbrechungs-Flussmarkierung zurückzieht. Keine Synchronisierung wird zwischen mehreren Threads implementiert als Reaktion auf eines der Ereignisse der vierten Gruppe.

[0092] Eine fünfte Gruppe von Ereignissen gemäß einem exemplarischen Ausführungsbeispiel ist spezifisch für die Architektur eines multithreading-fähigen Prozessors und wird gemäß dem beschriebenen Ausführungsbeispiel so implementiert, dass eine Reihe von Faktoren adressiert wird, die für eine multithreading-fähigen Prozessorumgebung speziell sind. Die fünfte Gruppe von Ereignissen umfasst ein Ereignis VIRTUAL NUKE (virtueller Nuke) **260**, ein Ereignis SYNCHRONIZATION (Synchronisierung) **262** und ein Ereignis SLEEP (Ruhen) **264**.

[0093] Das Ereignis VIRTUAL NUKE **260** ist ein Ereignis, das registriert wird in Bezug auf einen zweiten Thread, wenn (1) ein erster Thread in dem multithreading-fähigen Prozessor **30** ein anstehendes Ereignis aufweist (z. B. eines der vorstehend beschriebenen Ereignisse ist anstehend bzw. ausstehend), (2) der zweite Thread keine anstehenden Ereignisse aufweist (mit Ausnahme des Ereignisses **260**), und (3) ein Mikrobefehl mit entweder einer Flussmarkierung für eine gemeinsam genutzte Ressource **184** oder eine Synchronisierungs-Flussmarkierung **186** durch den Neuordnungspuffer **162** zurückgezogen wird. Ein Ereignis VIRTUAL NUKE **260** weist den Effekt des Aufrufens einer Behandlungsroutine für ein Virtual-Nuke-Ereignis auf, welche die Ausführung des zweiten Threads an dem Mikrobefehl nach dem zurückgezogenen Mikrobefehl mit den Flussmarkierungen **184** bzw. **186** neu startet.

[0094] Das Ereignis SYNCHRONIZATION (Synchronisierung) **262** wird durch Mikrocode signalisiert, wenn ein bestimmter Thread (z. B. ein erster Thread) erforderlich ist, um einen gemeinsam genutzten Zustand oder eine gemeinsam genutzte Ressource in dem multithreading-fähigen Prozessor **30** zu modifizieren. Zu diesem Zweck führt der Mikrocode-Sequencer **66** einen Synchronisierungs-Mikrobefehl in den Fluss bzw. Ablauf des ersten Threads ein und markiert den „Synchronisierungs-Mikrobefehl" zum Vermeiden einer Situation der gegenseitigen Blockierung sowohl mit einer Flussmarkierung **184** für eine gemeinsam genutzte Ressource und mit einer Synchronisierungs-Flussmarkierung **186**. Das Ereignis Synchronisierung **262** wird nur detektiert (oder registriert) nach dem Rückzug des Synchronisierungs-Mikrobefehls für den ersten Thread un nach dem Rückzug eines Mikrobefehls für den zweiten Thread mit einer zugeordneten Synchronisierungs-Flussmarkierung **186**. Ein Synchronisierungsereignis **262** weist den Effekt des Aufrufens einer Synchronisierungsereignis-Behandlungsroutine auf, welche die Ausführung des ersten Threads an einem Befehlszeiger neu startet, der in einem temporären Mikrocode-Register gespeichert ist. Weitere Einzelheiten zu der Behandlung eines Synchronisierungsereignisses **262** werden nachstehend im Text bereitgestellt. Der zweite Thread für ein VIRTUAL NUKE **260** aus.

[0095] Das Ereignis SLEEP (Ruhen) **264** ist ein Ereignis, das es bewirkt, dass ein relevanter Thread von einem aktiven Zustand in einen inaktiven Zustand (Ruhezustand) übergeht. Der inaktive Thread kann danach wieder einen Wechsel bzw. Übergang aus dem inaktiven in den aktiven Zustand vornehmen, durch ein entsprechend geeignetes Unterbrechungsereignis (BREAK). Die Beschaffenheit des Unterbrechungsereignisses, das den Thread zurück in den aktiven Zustand wechselt, ist abhängig von dem Ereignis SLEEP **264**, das den Thread in den inaktiven Zustand umgeschaltet hat. Der Eintrag in einen und aus einem aktiven Zustand durch die Threads wird nachstehend im Text näher beschrieben.

[0096] Die Abbildung aus Fig. 9 zeigt ein Blockdiagramm mit exemplarischen Inhalt der Neuordnungstabelle **180** in dem Neuordnungspuffer **162**, der nachstehend im Text beschrieben wird, um die Erkennung des Ereignisses und des Auslösepunkts (auch „Nuke-Punkt") in einem exemplarischen Ausführungsbeispiel der vorliegenden Erfindung zu be-

schreiben. Die Erkennung eines beliebigen der vorstehenden Ereignisse durch den Ereignisdetektor **188** in dem Block **222** kann als Reaktion darauf eintreten, dass das Ereignis **266** von einer internen Quelle in dem multithreading-fähigen Prozessor **30** oder von einer externen Quelle außerhalb des Prozessors **30** zu dem Ereignisdetektor **188** kommuniziert wird. Ein Beispiel für die derartige Kommunikation eines Ereignisses **266** kann ein Unterbrechungsvektor sein. Alternativ kann das Eintreten eines Ereignisses durch Fehlerinformationen **268** für einen Mikrobefehl eines bestimmten Threads (z. B. Thread 1) kommuniziert werden, der zurückgezogen wird und folglich durch den Rückzugszeiger **182** identifiziert wird. Hiermit wird festgestellt, dass für externe Ereignisse ein (1) Signal je Thread vorgesehen ist (z. B. die entsprechenden Signale **266** und **267**). Für interne Ereignisse gibt der Eintrag in dem Neuordnungspuffer **162**, der den Thread enthält, dem Thread vor, wen der Fehler betrifft, und zwar durch dessen Position (z. B. T0 ggü. T1). Nach dem Erkennen eines Ereignisses speichert der Ereignisdetektor **188** Ereignisinformationen (z. B. Ereignistyp, Ereignisquelle, etc.) in Bezug auf das jeweilige Ereignis in dem Ereignisinformationsregister **202** und registriert ferner ein anstehendes bzw. ausstehendes Ereignis für den relevanten Thread in dem Register **204** für anstehende Ereignisse. Wie dies vorstehend im Text beschrieben worden ist, umfasst die Registrierung eines anstehenden Ereignisses in dem Register **204** für anstehende Ereignisse für den relevanten Thread das Setzen eines Bits, das dem jeweiligen Ereignis zugeordnet ist, in dem Register **204**. Hiermit wird ferner festgestellt, dass das Ereignis effektiv detektiert werden kann durch Aktivierung eines Signals **211** für die Erkennung eines entsprechenden Ereignisses, wenn das Ereignis nicht durch das Setzen eines Bits in dem Ereignissperrregister **206** für den relevanten Thread gesperrt wird, und in einigen Fällen enthält ein Mikrobefehl eine entsprechende Flussmarkierung.

[0097] In erneutem Bezug auf das Flussdiagramm aus der Abbildung aus Fig. 7A unterbricht bzw. stoppt der Ereignisdetektor **188** nach der Erkennung des ersten Ereignisses für den ersten Thread in dem Block **222** den Rückzug des ersten Threads in dem Block **270** und aktiviert ein Signal "Pre-Nuke" **169**. Das Pre-Nuke-Signal **169** wird aktiviert, um eine Situation der gegenseitigen Blockierung zu vermeiden, in der der erste Thread die Befehls-Pipeline zum Ausschluss des zweiten Threads dominiert. Sollte im Besonderen der zweite Thread vom Zugriff auf die Befehls-Pipeline ausgeschlossen werden, können die Bedingungen in Bezug auf den zweiten Thread, die erforderlich sind, um eine multithreading-fähige Nuke-Operation zu beginnen, nicht auftreten. Das Pre-Nuke-Signal **169** wird entsprechend zu dem Front-End des Prozessors ausgebreitet und im Besonderen zu der Speicherausführungseinheit **42**, um die Prozessor-Pipeline in Bezug auf Mikrobefehle zu verarmen, die den ersten Thread bilden, für den das Ereignis detektiert worden ist. Die Verarmung der Prozessor-Pipeline kann zum Beispiel einfach ausgeführt werden, indem die Vorerfassung von Befehls- und SMC-Operationen (selbstmodifizierende Code-Operationen) deaktiviert wird, die von der Speicherausführungseinheit **42** oder anderen Komponenten des Front-End ausgeführt werden. Zusammengefasst wird durch einen Stopp der Anforderung bzw. Notwendigkeit von Mikrobefehlen des ersten Threads und/oder durch Anhalten oder substantielle Reduzierung der Zufuhr von Mikrobefehlen mit dem ersten Thread in die Prozessor-Pipeline dem zweiten Thread eine Präferenz in dem Prozessor erteilt, und ferner wird die Wahrscheinlichkeit für eine Situation der gegenseitigen Blockierung reduziert.

[0098] In dem Entscheidungskästchen **272** wird bestimmt, ob ein zweiter Thread aktiv ist in dem multithreading-fähigen Prozessor **30** und somit durch den Neuordnungspuffer **162** zurückgezogen wird. Wenn kein zweiter Thread aktiv ist, springt das Verfahren **220** direkt zu dem Block **274**, in dem eine erste Art von Löschoperation mit der Bezeichnung „Nuke-Operation" ausgeführt wird. Die Bestimmung, ob ein bestimmter Thread aktiv oder inaktiv ist, kann in Bezug auf die aktive Thread-Zustandsmaschine **171** ausgeführt werden, die von der Rückzugssteuerschaltung **168** verwaltet werden. Die Nuke-Operation beginnt mit der Aktivierung des Nuke-Signals **170**, das den Effekt des Löschens der Zustände sowohl des In-Order-Front-Ends als auch des Out-of-Order-Back-Ends des multithreading-fähigen Prozessors **30** aufweist, wie dies vorstehend im Text beschrieben worden ist. Da nur der erste Thread aktiv ist, muss der Effekt der Nuke-Operation auf die anderen Threads nicht berücksichtigt werden, die in dem multithreading-fähigen Prozessor **30** vorhanden und noch vorhanden sein können.

[0099] Wenn im anderen Fall in dem Entscheidungskästchen **272** bestimmt wird, dass ein zweiter Thread in dem multithreading-fähigen Prozessor **30** aktiv ist, so führt das Verfahren **220** mit der Ausführung einer Reihe von Operationen fort, die die Erkennung eines Löschpunkts (oder Nuke-Punkts) für den zweiten Thread darstellen, bei dem eine Nuke-Operation mit reduzierten negativen Folgen für den zweiten Thread ausgeführt werden kann. Die nach der Erkennung eines Löschpunkts ausgeführt Nuke-Operation ist die gleiche Operation, wie sie in dem Block **274** ausgeführt wird, und folglich löscht sie den Zustand des multithreading-fähigen Prozessors **30** (d. h. den Zustand für die ersten und zweiten Threads). Das Löschen des Zustands beinhaltet „Entleerungs"-Operationen von Mikrobefehlen, die an anderer Stelle in der Patentschrift beschrieben werden. In einem exemplarischen Ausführungsbeispiel, das in der vorliegenden Anmeldung offenbart wird, unter-

scheidet die nach dem Erkennen eines Lösch- bzw. Auslösepunkts ausgeführte Nuke-Operation nicht zwischen dem für einen ersten Thread verwalteten Zustand und dem für einen zweiten Thread verwalteten Zustand in dem multithreading-fähigen Prozessor **30**. In einem alternativen Ausführungsbeispiel kann die nach der Erkennung eines Löschpunkts ausgeführte Nuke-Operation den Zustand für nur einen einzigen Thread löschen (d. h. den Thread, für den das Ereignis detektiert worden ist), wobei ein signifikantes Maß der gemeinsamen Nutzung von Ressourcen in einem multithreading-fähigen Prozessor **30** auftritt, und wobei die genannten gemeinsam genutzten Ressourcen dynamisch aufgeteilt werden und deren Aufteilung aufgehoben wird, um mehrere Threads zu behandeln bzw. abzuarbeiten, wobei das Löschen des Zustands für einen einzigen Thread besonders komplex ist. Das vorliegende alternative Ausführungsbeispiel kann jedoch eine zunehmend komplexe Hardware erfordern.

[0100] Nach der positiven Bestimmung in dem Entscheidungskästchen **272** erfolgt in dem Entscheidungskästchen **278** eine weitere Bestimmung, ob in dem zweiten Thread ein Ereignis aufgetreten ist. Ein derartiges Ereignis kann jedes der vorstehend beschriebenen Ereignisse umfassen, mit Ausnahme des Ereignisses VIRTUAL NUKE **260**. Diese Bestimmung erfolgt wiederum durch den Ereignisdetektor **188**, der auf ein Ereignissignal **266** oder ein Fehlerinformationssignal **269** für den zweiten Thread anspricht. Informationen zu einem beliebigen Ereignis, mit dem der zweite Thread konfrontiert wird, wird in dem Abschnitt des Ereignisinformationsregisters **202** gespeichert, der dem zweiten Thread vorbehalten ist, und das Auftreten des Ereignisses wird in dem Register **204** für anstehende Ereignisse registriert.

[0101] Wenn der zweite Thread unabhängig mit einem Ereignis konfrontiert worden ist, so fährt das Verfahren direkt mit dem Block **280** fort, wobei eine multithreading-fähige Nuke-Operation ausgeführt wird, um den Zustand des multithreading-fähigen Prozessors **30** zu löschen. Sollte der zweite Thread alternativ mit keinem Ereignis konfrontiert worden sein, wird in dem Entscheidungskästchen **282** bestimmt, ob das erste Ereignis, auf das für den ersten Thread getroffen worden ist, die Modifikation eines gemeinsam genutzten Zustands oder gemeinsam genutzter Ressourcen erfordert, um das erste Ereignis zu behandeln. Wenn das erste Ereignis zum Beispiel ein Ereignis SYNCHRONIZATION (Synchronisierung) **262** gemäß der vorstehenden Beschreibung umfasst, zeigt dies an, dass der erste Thread den Zugriff auf eine gemeinsam genutzte Zustandsressource erfordert. Das Synchronisierungsereignis **262** kann identifiziert werden durch den Rückzug eines Synchronisierungs-Mikrobefehls für den ersten Thread, der zugeordnete Flussmarkierungen **184** und **186** für gemeinsam genutzte Ressourcen und Synchronisierung aufweist. Die Abbildung aus Fig. 10 zeigt ein Blockdiagramm, ähnlich dem Blockdiagramm aus Fig. 9, das exemplarischen Inhalt für die Neuordnungstabelle **180** zeigt. Der dem ersten Thread (z. B. Thread 0) zugeordnete Abschnitt der Tabelle **180** umfasst gemäß der Abbildung einen Synchronisierungs-Mikrobefehl, auf den durch den Rückzungszeiger **182** verwiesen wird. Die Synchronisierungs-Mikrobefehle sind ferner mit einer zugeordneten Flussmarkierung **184** für gemeinsam genutzte Ressourcen und mit einer Synchronisierungs-Flussmarkierung **186** dargestellt. Der Rückzug des veranschaulichten Synchronisierungs-Mikrobefehls wird von dem Ereignisdetektor **188** als das Auftreten eines Synchronisierungsereignisses **262** registriert.

[0102] Wenn für das erste Ereignis für den ersten Thread (z. B. Thread 0) bestimmt wird, dass ein gemeinsam genutzter Zustand oder eine gemeinsam genutzte Ressource nicht modifiziert wird, fährt das Verfahren **220** mit dem Entscheidungskästchen **284** fort, in dem bestimmt wird, ob der der zweite Thread (z. B. Thread 1) einen Mikrobefehl zurückzieht, der eine zugeordnete Flussmarkierung **184** für gemeinsam genutzte Ressourcen aufweist. In Bezug auf die Abbildung aus Fig. 9 ist der Rückzugszeiger **182** für den Thread 1 dargestellt mit einem Verweis auf einen Mikrobefehl mit einer Flussmarkierung **184** für gemeinsam genutzte Ressourcen und einer Synchronisierungs-Flussmarkierung **186**. In dieser Situation ist die Bedingung aus dem Entscheidungskästchen **284** erfüllt, und das Verfahren **220** fährt entsprechend mit dem Block **280** fort, in dem die multithreading-fähige Nuke-Operation ausgeführt wird. Sollte der Rückzugszeiger **182** für den zweiten Thread (z. B. Thread 1) alternativ nicht auf einen Mikrobefehl verweisen, der entweder eine Flussmarkierung **184** für gemeinsam genutzte Ressourcen oder eine Synchronisierungs-Flussmarkierung **186** aufweist, fährt das Verfahren mit dem Block **286** fort, in dem der Rückzug des zweiten Threads durch den Fortschritt des Rückzugszeigers **182** andauert. Von dem Block **286** springt das Verfahren **220** zurück zu dem Entscheidungskästchen **278**, in dem wiederum bestimmt wird, ob der zweite Thread auf ein Ereignis getroffen ist.

[0103] Wenn in dem Entscheidungskästchen **282** bestimmt wird, dass die Behandlung des ersten Ereignisses für den ersten Thread (z. B. Thread 0) die Modifikation einer Ressource mit gemeinsam genutzten Zustand erfordert, fährt das Verfahren **220** mit dem Entscheidungskästchen **288** fort, in dem bestimmt wird, ob der zweite Thread (z. B. Thread 1) einen Mikrobefehl zurückzieht, der eine zugeordnete Synchronisierungs-Flussmarkierung **186** aufweist. Wenn dies der Fall ist, wird die multithreading-fähige Nuke-Operation in dem Block **280** ausgeführt. Wenn dies nicht der Fall ist, dauert der Rückzug des Mikrobefehls für den zweiten Thread in Block **286** an, bis

entweder auf ein Ereignis für den zweiten Thread getroffen wird oder bis der Rückzugszeiger **182** für den zweiten Thread einen Mikrobefehl mit einer zugeordneten Synchronisierungs-Flussmarkierung **186** indexiert.

[0104] Nach dem Beginn der Nuke-Operation in dem Block **280** fährt in dem Block **290** eine entsprechende Ereignisbehandlungsroutine **67**, die in Mikrocode implementiert und von dem Mikrocode-Sequencer **66** ablaufgesteuert wird,, mit der Behandlung des relevanten Ereignisses fort.

Ereignis Virtual Nuke

[0105] Wie dies bereits vorstehend im Text beschrieben worden ist, wird das Ereignis VIRTUAL NUKE **260** etwas anders behandelt als andere Ereignisse. Zu diesem Zweck zeigt die Abbildung aus Fig. 7B ein Flussdiagramm, das ein Verfahren **291** gemäß einem exemplarischen Ausführungsbeispiel zum Detektieren und Behandeln eines Ereignisses VIRTUAL NUKE **260** veranschaulicht. Das Verfahren **291** nimmt an, dass aktuell keine Ereignisse für einen zweiten Thread anstehen (d. h. in einem ausstehenden Register für den zweiten Thread aufgezeichnet sind).

[0106] Das Verfahren **291** beginnt in dem Block **292** mit der Erkennung eines ersten Ereignisses für den ersten Thread durch den Ereignisdetektor **188**. Bei einem derartigen Ereignis kann es sich um jedes der vorstehend in Bezug auf die Abbildung aus Fig. 8 beschriebenen Ereignisse handeln.

[0107] In dem Block **293** stoppt der Ereignisdetektor **188** den Rückzug des ersten Threads. In dem Block **294** detektiert der Ereignisdetektor **188** den Rückzug eines ersten Mikrobefehls mit entweder einer Flussmarkierung **184** für eine gemeinsam genutzte Ressource oder einer Synchronisierungs-Flussmarkierung. In dem Block **295** wird eine Behandlungsroutine "Virtual Nuke" von dem Mikrocode-Sequencer **66** aufgerufen. Die Ereignisbehandlungsroutine "Virtual Nuke" beginnt in dem Block **296** erneut mit der Ausführung des zweiten Threads bei einem Mikrobefehl nach dem vorstehend in Block **294** zurückgezogenen Mikrobefehl. Das Verfahren **291** endet in dem Block **297**.

Die Nuke-Operation

[0108] Die Abbildung aus Fig. 11A zeigt ein Flussdiagramm, das ein Verfahren **300** gemäß einem exemplarischen Ausführungsbeispiel veranschaulicht, zur Ausführung einer Auslöseoperation (oder Nuke-Operation) in einem multithreading-fähigen Prozessor, der mindestens erste und zweite Threads unterstützt. Das Verfahren **300** beginnt mit dem Block **302** mit der Aktivierung des Nuke-Signals **170** durch den Ereignisdetektor **188** als Reaktion auf das Auftreten und die Erkennung eines Ereignisses. Das Nuke-Signal **170** wird an zahlreiche funktionale Einheiten in dem multithreading-fähigen Prozessor **30** kommuniziert, und die Aktivierung und Deaktivierung des Signals definiert ein Fenster, in dem Aktivitäten zur Vorbereitung des Löschens bzw. Auslösens des Zustands und der Konfiguration von funktionalen Einheiten ausgeführt werden. Die Abbildung aus Fig. 12 zeigt ein Takt- bzw. Zeitsteuerungsdiagramm, das zeigt, dass die Aktivierung des Nuke-Signals **170** synchron zu der ansteigenden Flanke eines Taktsignals **304** auftritt.

[0109] In dem Block **303** wird die aktive Thread-Zustandsmaschine beurteilt bzw. evaluiert.

[0110] In dem Block **306** werden die Sequenznummer und das letzte Mikrobefehlssignal, das anzeigt, ob der Mikrobefehl, bei dem das Ereignis auftritt, zurückgezogen worden ist oder nicht, sowohl für die ersten als auch für die zweiten Threads zu der Zuordnungs- und freien Listenverwaltungslogik **122** und der TBIT kommuniziert, wobei letztere eine Struktur in einer Trace Branch Prediction Unit (TBPU) darstellt (die wiederum Bestandteil von TDE **60** ist), um Makrobefehls- und Mikrobefehls-Zeigerinformationen in dem In-Order-Front-End des Prozessors **30** nachzuverfolgen. Der TBIT nutzt diese Informationen zum Zwischenspeichern von Informationen zu dem Ereignis (z. B. des Mikrobefehls- und Makrobefehls-Befehlszeigers).

[0111] In dem Block **308** konstruiert und verteilt der Ereignisdetektor **188** einen Ereignisvektor für jeden der ersten und zweiten Threads an den Mikrocode-Sequencer **66**. Jeder Ereignisvektor beinhaltet unter anderem Informationen, die folgendes identifizieren: (1) die Position des physikalischen Neuordnungspuffers, die zurückgezogen worden ist, als der Nuke-Punkt (oder Auslösepunkt) lokalisiert wurde (d. h. der Wert jedes Rückzugszeigers **182**, wenn der Nuke-Punkt identifiziert worden ist); (2) einen Bezeichner für die Ereignisbehandlungsroutine, welcher eine Position in dem Mikrocode-Sequencer **66** identifiziert, an der sich Mikrocode, der eine Ereignisbehandlungsroutine **67** für die Verarbeitung des detektierten Ereignisses, befindet; und (3) einen Thread-Bezeichner zur Identifikation entweder des ersten oder des zweiten Threads; und (4) ein Thread-Prioritätsbit, das die Priorität der Ereignisbehandlungsroutine **67** im Verhältnis zu der Ereignisbehandlungsroutine bestimmt, die für andere Threads aufgerufen worden ist.

[0112] In dem Block **310** verwendet die Zuordnungs- und freie Listenverwaltungslogik **1232** die in dem Block **306** kommunizierten Sequenznummern zum Fortschalten einer Schattenregister-Alias-Tabelle (Schatten-RAT) an einen Punkt an dem der Nu-

ke-Punkt detektiert worden ist, und in dem Block **312** wird der Zustand der primären Register-Alias-Tabelle **1209** aus der Schattenregister-Alias-Tabelle wiederhergestellt.

[0113] In dem Block **314** stellt die Zuordnungs- und freie Listenverwaltungslogik **122** Registernummern (oder „Marbles") aus dem freien Listenmanager **134** wieder her und ordnet die wiederhergestellten Registernummern der Trash-Heap-Array **132** zu, von der die Registernummern wiederum zugeordnet werden können. Die Zuordnungs- und freie Listenverwaltungslogik **122** aktiviert bzw. behauptet ferner ein „wiederhergestelltes" Signal (nicht abgebildet), wenn alle entsprechenden Registernummern aus dem freien Listenmanager **134** wiederhergestellt worden sind. Das Nuke-Signal **170** wird in einem aktivierten Zustand gehalten, bis das „wiederhergestellte" Signal von der Zuordnungs- und freien Listenverwaltungslogik **122** empfangen wird.

[0114] In dem Block **316** werden alle „älteren" Speicherungen (d. h. Speicherungen, die zurückgezogen worden sind, jedoch im Speicher noch nicht aktualisiert worden sind) sowohl für die ersten als auch für die zweiten Threads aus dem Speicheranordnungspuffer unter Verwendung einer Speicherwegschreibelogik (nicht abgebildet) entleert.

[0115] In dem Block **320** deaktiviert der Ereignisdetektor **188** danach das Nuke-Signal **170** an der ansteigenden Flanke des Taktsignals **304**, wie dies in der Abbildung aus Fig. 12 dargestellt ist. Hiermit wird festgestellt, dass das Nuke-Signal **170** über mindestens drei Taktzyklen des Taktsignals **304** in einem aktivierten Zustand gehalten worden ist. Für den Fall, dass das „wiederhergestellte" Signal von der Zuordnungs- und freien Listenverwaltungslogik **122** nicht innerhalb der ersten beiden Taktzyklen des Taktsignals **304** nach der Aktivierung des Nuke-Signals **170** wiederhergestellt wird, erweitert bzw. verlängert der Ereignisdetektor **188** die Aktivierung des Nuke-Signals **170** über die veranschaulichten drei Taktzyklen hinaus. Das Nuke-Signal **170** kann in einem Ausführungsbeispiel ausreichend lang gehalten werden (z. B. über die drei Taktzyklen), um die Ausführung der vorstehend beschriebenen Blöcke **303**, **306** und **308** zu ermöglichen. Es kann erforderlich sein, dass das Nuke-Signal **170** über zusätzliche Zyklen gehalten wird, um eine Ausführung der Blöcke **310**, **312**, **314** und **316** zu ermöglichen. Zu diesem Zweck aktiviert der Speicheranordnungs- bzw. Speicherreihenfolgepuffer ein Signal „Speicherpuffer geleert", um die Aktivierung des Nuke-Signals zu erweitern.

[0116] In dem Block **322** prüfen der Mikrocode-Sequencer **66** und andere funktionale Einheiten in dem multithreading-fähigen Prozessor **30** die „aktiven Bits", die von der aktiven Thread-Zustandsmaschine **171** gehalten werden, um zu bestimmen, ob sich die ersten und zweiten Threads jeweils nach dem Auftreten des Ereignisses in einem aktiven oder einem inaktiven Zustand befinden. Im Besonderen verwaltet die aktive Thread-Zustandsmaschine **171** eine entsprechend Bit-Anzeige für jeden Thread, der noch in dem multithreading-fähigen Prozessor **30** vorhanden ist, die anzeigt, ob sich der relevante Thread in einem aktiven oder inaktiven (Ruhezustand) Zustand befindet. Das von dem Ereignisdetektor **188** detektierte Ereignis, als Reaktion auf welches der Ereignisdetektor **188** das Nuke-Signal **170** aktiviert hat, kann ein Ereignis SLEEP (Ruhezustandsereignis) **264** oder ein Ereignis BREAK (Unterbrechungsereignis) umfassen, das entweder den ersten oder den zweiten Thread zwischen aktiven und inaktiven Zustand umschaltet. Wie dies in der Abbildung aus Fig. 12 unter **324** dargestellt ist, wird die aktive Thread-Zustandsmaschine **171** während der Aktivierung des Nuke-Signals **170** evaluiert, und der Zustand der „aktiven Bits" gilt entsprechend als gültig nach der Deaktivierung des Nuke-Signals **170**.

[0117] In dem Entscheidungskästchen **326** bestimmt jede der funktionalen Einheiten, die die aktiven Bits der aktiven Thread-Zustandsmaschine **171** geprüft bzw. untersucht hat, ob die ersten und zweiten Threads aktiv sind. Wenn auf der Basis des Zustands der aktiven Bits bestimmt wird, dass beide Threads aktiv sind, fährt das Verfahren **300** mit dem Block **328** fort, in dem jede der funktionalen Einheiten so konfiguriert wird, dass die aktiven ersten und zweiten Threads beide unterstützt und behandelt werden. Zum Beispiel können die in Verbindung mit verschiedenen funktionalen Einheiten bereitgestellte Speicher- und Pufferfunktionalitäten logisch aufgeteilt werden durch Aktivierung eines zweiten Zeigers oder einer zweiten Gruppe von Zeigern, die auf eine bestimmte Gruppe (oder einen Bereich) von Einträgen in einer Speicheranordnung begrenzt sind. Ferner kann eine bestimmte MT-spezifische Unterstützung aktiviert werden, wenn zwei Threads aktiv sind. Zum Beispiel kann die Thread-Auswahllogik, die dem Mikrocode-Sequencer zugeordnet ist, Threads von einem ersten Thread (z. B. T0), von einem zweiten Thread (z. B. T1) oder sowohl von einem ersten als auch einem zweiten Thread (z. B. T0 und T1) im „Ping-Pong"-System ablaufgesteuert werden, und zwar auf der Basis der aktiven Thread-Zustandsmaschine **171**. Ferner kann eine lokalisierte Takttorsteuerung auf der Basis der Bitausgabe der aktiven Thread-Zustandsmaschine ausgeführt werden. In einem weiteren Ausführungsbeispiel kann jede Anzahl von Zustandsmaschinen in einem Prozessor ihr Verhalten modifizieren oder den Zustand verändern auf der Basis der Ausgabe der aktiven Thread-Zustandsmaschine. In dem Block **330** fährt der Mikrocode-Sequencer **66** danach mit der Ablaufsteuerung von Mikrobefehlen sowohl für die ersten als auch die zweiten Threads fort.

[0118] Wenn in dem Entscheidungskästchen **326** ferner bestimmt wird, dass nur einer der ersten und zweiten Threads aktiv ist, oder dass beide Threads inaktiv sind, so wird jede der funktionalen Einheiten so konfiguriert, dass nur ein einziger aktiver Thread in dem Block **332** unterstützt und behandelt wird, und wobei eine gewisse MT-spezifische Unterstützung deaktiviert werden kann. Wenn keine Threads aktiv sind, sind die funktionalen Einheiten als Standardeinstellung so konfiguriert, dass ein einzelner aktiver Thread unterstützt wird. Für den Fall, dass eine funktionale Einheit vorher so konfiguriert wurde (z. B. logisch aufgeteilt), dass mehrere Threads unterstützt werden, so können die Zeiger, die zur Unterstützung weiterer Threads verwendet werden, deaktiviert werden, und die Reihe der Einträge in einer Datenanordnung, auf die der verbliebene Zeiger Bezug nehmen kann, kann erweitert werden, so dass sie Einträge enthält, auf die vorher durch die deaktivierten Zeiger verwiesen wurde. Auf diese Weise wird hiermit festgestellt, dass Dateneinträge, die vorher anderen Threads zugeordnet gewesen sind, danach zur Verwendung durch einen aktiven Thread zur Verfügung gestellt werden können. Dadurch, dass mehr Ressourcen für den einzelnen aktiven Threads zur Verfügung stehen, wenn weitere Threads inaktiv sind, kann die Performance des einzelnen verbliebenen Threads im Verhältnis zu dessen Performance verbessert werden, wenn auch andere bzw. weitere Threads in dem multithreading-fähigen Prozessor **30** unterstützt werden.

[0119] In dem Block **334** ignoriert der Mikrocode-Sequencer **66** Ereignisvektoren für einen inaktiven Thread oder inaktive Threads und führt eine Ablaufsteuerung von Mikrobefehlen nur für einen möglichen aktiven Thread aus. Wenn keine Threads aktiv sind, ignoriert der Mikrocode-Sequencer **66** die Ereignisvektoren für alle Threads.

[0120] Durch die Bereitstellung aktiver Bits, die von der aktiven Thread-Zustandsmaschine **171** verwaltet werden, die von verschiedenen funktionalen Einheiten nach der Deaktivierung des Nuke-Signals **170** geprüft werden (das das Ende einer Nuke-Operation anzeigt), wird eine praktische und zentrale Anzeige bereitgestellt, gemäß der die verschiedenen funktionalen Einheiten konfiguriert werden können, um eine korrekte Anzahl von aktiven Threads in einem multithreading-fähigen Prozessor **30** nach Abschluss der Nuke-Operation zu unterstützen.

[0121] Die Abbildung aus Fig. 11B zeigt ein Blockdiagramm, das eine exemplarische Konfigurationslogik **329** darstellt, die einer funktionalen Einheit **331** zugeordnet ist und die so arbeitet, dass sie die funktionale Einheit **331** so konfiguriert, dass ein oder mehrere aktive Threads in dem multithreading-fähigen Prozessor unterstützt werden. Bei der funktionalen Einheit **331** kann es sich um jede der vorstehend beschriebenen funktionalen Einheiten handeln oder um jede andere funktionale Einheit, für welche dem Fachmann auf dem Gebiet die Integration in einem Prozessor schlüssig erscheint. Gemäß der Abbildung weist die funktionale Einheit **331** Speicherungs- und Logikkomponenten auf, die von der Konfigurationslogik **329** konfiguriert werden. Zum Beispiel kann die Speicherkomponente eine Sammlung von Registern umfassen. Jedes dieser Register kann dem Speichern eines Mikrobefehls oder von Daten für einen bestimmten dieser Threads zugeordnet werden, wenn mehrere Threads aktiv sind (d. h. wenn ein Prozessor in einem MT-Modus arbeitet). Demgemäß ist die Speicherkomponente gemäß der Abbildung aus Fig. 11B logisch so aufgeteilt dargestellt, dass sie erste und zweite Threads (z. B. T0 und T1) unterstützt. Natürlich kann die Speicherkomponente auch so aufgeteilt werden, dass jede beliebige Anzahl von aktiven Threads unterstützt wird.

[0122] Die Logikkomponente weist gemäß der Abbildung eine MT-Logik auf, die speziell der Unterstützung des multithreading-fähigen Betriebs in dem Prozessor dient (d. h. einem MT-Modus).

[0123] Die Konfigurationslogik **329** ist so dargestellt, dass sie die Zeigerwerte **333** erhält, die an die Speicherkomponente der funktionalen Einheit **331** ausgegeben werden. In einem exemplarischen Ausführungsbeispiel werden diese Zeigerwerte **333** eingesetzt, um die Speicherkomponente logisch aufzuteilen. Zum Beispiel kann ein separates Paar von Lese- und Schreib-Zeigerwerten für jeden aktiven Thread erzeugt werden. Die oberen und unteren Begrenzungen der Zeigerwerte für jeden Thread werden durch die Konfigurationslogik **329** abhängig von der Anzahl der aktiven Threads bestimmt. Zum Beispiel kann der Bereich der Register, die durch eine Reihe von Zeigerwerten für einen bestimmten Thread angezeigt werden, vergrößert werden, um die Register abzudecken, die vorher einem anderen Thread zugeordnet waren, sollte der andere Thread inaktiv werden.

[0124] Die Konfigurationslogik **329** weist ferner Anzeigen **335** zur Freigabe der MT-Unterstützung auf, die an die Logikkomponente der funktionalen Einheit ausgegeben worden sind, um die MT-Unterstützungslogik der funktionalen Logik **331** zu aktivieren oder zu deaktivieren.

[0125] Die aktiven Bits **327**, die von der aktiven Thread-Zustandsmaschine **174** ausgegeben werden, stellen eine Eingabe in die Konfigurationslogik bereit und werden von der Konfigurationslogik **329** zur Erzeugung der entsprechenden Zeigerwerte **333** und zur Bereitstellung der entsprechenden Ausgaben zur Freigabe bzw. Aktivierung der MT-Unterstützung bereitgestellt.

Exklusiver Zugriff durch eine Ereignisbehandlungsroutine

[0126] Bestimmte Ereignisbehandlungsroutinen (z. B. die Routinen für die Behandlung von Paging- und Synchronisierungsereignissen) erfordern einen exklusiven Zugriff auf den multithreading-fähigen Prozessor **30** zur Nutzung gemeinsam genutzter Ressourcen und zum Modifizieren des gemeinsam genutzten Zustands. Demgemäß implementiert der Mikrocode-Sequencer **66** eine exklusive Zugriffs-Zustandsmaschine **69**, die nacheinander den Ereignisbehandlungsroutinen für die ersten und zweiten Threads exklusiven Zugriff gewährt, wenn eine dieser Ereignisbehandlungsroutinen einen solchen exklusiven Zugriff benötigt. Die exklusive Zugriffs-Zustandsmaschine **69** kann nur referenziert werden, wenn mehr als ein Thread in dem multithreading-fähigen Prozessor **30** aktiv ist. Eine Flussmarkierung, die einer Ereignisbehandlungsroutine zugeordnet ist, die mit exklusivem Zugriff bereitgestellt ist, wird in den Fluss für den Thread eingefügt, um das Ende des exklusiven Codes zu markieren, die Ereignisbehandlungsroutine umfassend. Sobald der exklusive Zugriff für alle Threads abgeschlossen ist, nimmt der Mikrocode-Sequencer **66** wieder die normale Ausgabe von Mikrobefehlen auf.

[0127] Die Abbilddung aus Fig. 13 zeigt ein Flussdiagramm, das ein Verfahren **400** gemäß einem exemplarischen Ausführungsbeispiel veranschaulicht, und zwar des Bereitstellens eines exklusiven Zugriffs für eine Ereignisbehandlungsroutine **67** in einem multithreading-fähigen Prozessor **30**. Das Verfahren **400** beginnt mit dem Block **402** mit dem Empfang erster und zweiter Ereignisvektoren durch den Mikrocode-Sequencer **66** für die entsprechenden ersten und zweiten Threads von dem Ereignisdetektor **188**. Wie dies vorstehend beschrieben worden ist identifiziert jeder der ersten und zweiten Ereignisvektoren eine entsprechende Ereignisbehandlungsroutine **67**.

[0128] In dem Entscheidungskästchen **403** wird bestimmt, ob mehr als ein (1) Thread aktiv ist. Diese Bestimmung erfolgt durch den Mikrocode-Sequencer in Bezug auf die aktive Thread-Zustandsmaschine **171**. Wenn dies nicht der Fall ist, fährt das Verfahren **400** mit dem Block **434** fort. Wenn dies der Fall ist, fährt das Verfahren mit dem Entscheidungskästchen **404** fort.

[0129] In dem Entscheidungskästchen **404** bestimmt der Mikrocode-Sequencer **66**, ob die erste oder die zweite Ereignisbehandlungsroutine **67** einen exklusiven Zugriff auf eine gemeinsam genutzte Ressource benötigt oder einen gemeinsam genutzten Zustand modifiziert. Wenn dies der Fall ist, implementiert der Block **406** des Mikrocode-Sequencers **66** die exklusive Zugriffs-Zustandsmaschine **69**, um nacheinander einen exklusiven Zugriff auf jede der ersten und zweiten Ereignisbehandlungsroutinen **67** zu gewähren. Die Abbildung aus Fig. 14 zeigt ein Zustandsdiagramm, das den Betrieb der exklusiven Zugriffs-Zustandsmaschine **69** gemäß einem exemplarischen Ausführungsbeispiel darstellt. Die Zustandsmaschine weist gemäß der Abbildung fünf Zustände auf. In einem ersten Zustand **408** wird der Mikrocode für den ersten Thread und für den zweiten Thread durch den Mikrocode-Sequencer **66** ausgegeben. Beim Auftreten einer Nuke-Operation **410** als Reaktion auf ein Ereignis, das eine exklusive Zugriffs-Ereignisbehandlungsroutine erfordert, wechselt die Zustandsmaschine **69** in einen zweiten Zustand **412**, in dem eine erste Ereignisbehandlungsroutine **67** (d. h. Mikrobefehle) ausgegeben wird, die einem Ereignis für einen ersten Thread zugeordnet ist. Nach der Ablaufsteuerung aller Mikrobefehle, welche die erste Ereignisbehandlungsroutine **67** darstellen und ferner nach Abschluss aller Operationen, die durch diese Mikrobefehle angewiesen worden sind, gibt der Mikrocode-Sequencer **66** danach einen Anhalte-Mikrobefehl (z. B. weist der Mikrobefehl eine zugeordnete Anhalte-Flussmarkierung auf) bei **414** aus, um die Zustandsmaschine **69** aus dem zweiten Zustand **412** in einen dritten Zustand **416** umzuschalten, in dem die Ausgabe der Mikrobefehle eines ersten Threads angehalten wird. Bei **418** wird der bei **414** ausgegebene Anhalte-Mikrobefehl aus dem Neuordnungspuffer **162** zurückgezogen, um dadurch die Zustandsmaschine **69** aus dem dritten Zustand **416** in einen vierten Zustand **420** umzuschalten, in dem der Mikrocode-Sequencer **66** die zweite Ereignisbehandlungsroutine **67** ausgibt, die einem Ereignis für den zweiten Thread zugeordnet ist. Nach der Ablaufsteuerung aller Mikrobefehle, welche die zweite Ereignisbehandlungsroutine **67** bilden, sowie nach Abschluss aller durch diese Mikrobefehle angewiesenen Operationen, gibt der Mikrocode-Sequencer **66** bei **422** einen weiteren Anhalte-Mikrobefehl aus, um die Zustandsmaschine **69** aus dem vierten Zustand in einen fünften Zustand **424** umzuschalten, in dem die zweite Ereignisbehandlungsroutine **67** angehalten wird. Bei **426** wird der bei **422** ausgegebene Mikrobefehl aus dem Neuordnungspuffer **162** zurückgezogen, um dadurch die Zustandsmaschine **69** aus dem fünften Zustand **424** zurück in den ersten Zustand **408** umzuschalten.

[0130] In dem Block **432** wird die normale Ablaufsteuerung und Ausgabe von Mikrobefehlen für den ersten und den zweiten Thread wieder aufgenommen, wobei angenommen wird, dass beide Threads aktiv sind.

[0131] Wenn in dem Entscheidungskästchen **404** alternativ bestimmt wird, dass keine der ersten und zweiten Ereignisbehandlungsroutinen einen exklusiven Zugriff auf gemeinsam genutzte Ressourcen oder gemeinsam genutzten Zustand des Prozessors **30** erfordert, so fährt das Verfahren mit dem Block

434 fort, in dem der Mikrocode-Sequencer **66** eine Ablaufsteuerung des Mikrocodes vornimmt, der die ersten und zweiten Ereignisbehandlungsroutinen **67** bildet, und zwar auf nicht-exklusive, verschachtelte Art und Weise.

Die aktive Thread-Zustandsmaschine (**171**)

[0132] Die Abbildung aus Fig. 15 zeigt ein Zustandsdiagramm **500**, das Zustände gemäß einem exemplarischen Ausführungsbeispiel veranschaulicht, die belegt werden können durch die aktive Thread-Zustandsmaschine **171**, und wobei die Abbildung ferner Übergangsereignisse gemäß einem exemplarischen Ausführungsbeispiel veranschaulicht, die einen Übergang der aktiven Thread-Zustandsmaschine **171** zwischen den verschiedenen Zuständen bewirken können.

[0133] Die aktive Thread-Zustandsmaschine **171** befindet sich in der Abbildung in einem von vier Zuständen, nämlich einem Zustand **502** Einzel-Thread 0 (ST0), einem Zustand **504** Einzel-Thread 1 (ST1), einem Zustand **506** Multi-Thread (MT) und einem Zustand **508** kein Thread (ZT). Die aktive Thread-Zustandsmaschine **171** verwaltet ein einzelnes aktives Bit für jeden Thread, das, wenn es gesetzt ist, den zugeordneten Thread als aktiv identifiziert, und wobei es, wenn es zurückgesetzt ist, anzeigt, dass der zugeordnete Thread inaktiv ist bzw. sich im Ruhezustand befindet.

[0134] Die Übergänge zwischen den vier Zuständen **502**–**508** werden durch Ereignispaare ausgelöst, wobei jedes Ereignis eines Ereignispaares den ersten oder den zweiten Thread betrifft. In dem Zustandsdiagramm **500** ist eine Reihe von Ereignisarten als zu einem Übergang zwischen den Zuständen beitragend angezeigt. Im Besonderen handelt es sich bei dem Ereignis SLEEP (Ruhen) um ein Ereignis, dass bewirkt, dass ein Thread inaktiv wird. Ein Ereignis BREAK (Unterbrechung) ist ein Ereignis, das, wenn es für einen bestimmten Thread auftritt, bewirkt, dass der Thread aus einem inaktiven Zustand in einen aktiven Zustand wechselt. Ob sich ein bestimmtes Ereignis als ein Ereignis BREAK eignet, kann abhängig sein von dem Ereignis SLEEP, das bewirkt hat, dass der Thread inaktiv wird. Im Besonderen bewirken nur bestimmte Ereignisse, dass ein Thread aktiv wird, wenn er einmal inaktiv ist, als Folge eines speziellen Ruhezustandsereignisses. Ein Ereignis NUKE ist jedes Ereignis, das wenn es für einen bestimmten Thread auftritt, zur Ausführung einer Nuke-Operation führt, wie dies bereits vorstehend im Text beschrieben worden ist. Alle vorstehend im Text in Bezug auf die Abbildung aus Fig. 8 beschriebenen Ereignisse umfassen potenziell Nuke-Ereignisse. Schließlich wird in dem Zustandsdiagramm **500** das Auftreten von „keinem Ereignis" in Bezug auf einen bestimmten Thread als ein Zustand angezeigt, der in Kombination mit dem Auftreten eines Ereignisses in Bezug auf einen weiteren Thread vorhanden sein kann, um einen Zustandswechsel zu bewirken.

[0135] Wenn in einem Ausführungsbeispiel ein Ereignis SLEEP für einen bestimmten Thread signalisiert wird und ein Ereignis BREAK für diesen Thread ansteht, so wird das Ereignis BREAK sofort behandelt bzw. bearbeitet (z. B. der Thread wechselt nicht in den Ruhezustand und wacht später wieder auf, um das Ereignis BREAK zu behandeln). Das umgekehrte Phänomen kann ebenfalls gelten, wobei ein Ereignis BREAK für einen bestimmten Thread signalisiert werden kann, und wobei ein Ereignis SLEEP ansteht, woraufhin das Ereignis BREAK bearbeitet wird.

[0136] Nach der Aktivierung des Nuke-Signals **170** durch den Ereignisdetektor **188** wird die aktive Thread-Zustandsmaschine **171** evaluiert, wie dies unter **324** in Fig. 12 dargestellt wird. Nach der Deaktivierung des Nuke-Signals **170** werden alle funktionalen Einheiten in dem multithreading-fähigen Prozessor **30** auf der Basis der aktiven Bits konfiguriert, die durch die aktive Thread-Zustandsmaschine **171** verwaltet werden. Im Besonderen breitet die Prüf-, Wiedergabe- und Rückzugseinheit (CRU) **160** ein auf der Basis der aktiven Bits erzeugtes Signal an alle betroffenen funktionalen Einheiten aus, um den funktionalen Einheiten anzuzeigen, wie viele Threads noch in dem multithreading-fähigen Prozessor existieren, und welche dieser Threads aktiv sind. Nach der Aktvierung des Nuke-Signals **170** wird die Konfiguration der funktionalen Einheiten (z. B. Aufteilung bzw. Partitionierung oder Aufhebung der Aufteilung) für gewöhnlich in einem Taktzyklus des Taktsignals **304** abgeschlossen.

Verlassen eines Threads und Eintreten in einen Thread

[0137] Die vorliegende Erfindung schlägt einen beispielhaften Mechanismus vor, durch den Threads in einem multithreading-fähigen Prozessor **30** eintreten und austreten können (z. B. aktiv oder inaktiv werden können), wobei ein derartiges Eintreten und Austreten in einer einheitlichen Folge bzw. Sequenz unabhängig von der Anzahl der aktiven Threads erfolgt, und wobei Taktsignale an verschiedene funktionale Einheiten problemlos angehalten werden können, wenn keine weiteren Threads in dem multithreading-fähigen Prozessor **30** aktiv sind bzw. laufen.

[0138] Wie dies bereits vorstehend im Text in Bezug auf das Zustandsdiagramm **500** beschrieben worden ist, erfolgt das Eintreten in einen Thread (oder eine Aktivierung) als Reaktion auf die Erkennung eines Ereignisses BREAK für einen aktuell inaktiven Thread. Die Definition eines Ereignisses BREAK für einen bestimmten inaktiven Thread ist abhängig von dem Grund dafür, dass der relevante Thread inaktiv ist.

Das Austreten aus einem Thread erfolgt als Reaktion auf ein Ereignis SLEEP für einen aktuell aktiven Thread. Zu den Beispielen der Ereignisse SLEEP zählen die Ausführung eines Befehls Anhalten bzw. Stopp (HLT), der in einem aktiven Thread enthalten ist, das Erkennen eines Zustands SHUTDOWN (Ausschaltzustand) oder eines Zustands ERROR SHUTDOWN oder eines Zustands "wait for SIPI" bzw. "warten auf SIPI" (SIPI als englische Abkürzung von Start-Up Inter-Processor Interrupt) in Bezug auf den aktiven Thread.

[0139] Die Abbildung aus Fig. 16A zeigt ein Flussdiagramm eines Verfahrens **600** gemäß einem exemplarischen Ausführungsbeispiel der vorliegenden Erfindung des Austretens aus einem aktiven Thread beim Erkennen eines Ereignisses SLEEP (Ruhen) für den aktiven Thread. Das Verfahren **600** beginnt mit dem Block **602**, in dem der gesamte erforderte Zustand für den aktiven Thread gespeichert wird, und in dem für alle Registereinträge in der Registerdatei **124**, die vor Mikrobefehlen für den aktiven Thread zugeordnet worden sind, die Zuordnung aufgehoben wird. Als reines Beispiel werden von den 128 Registereinträgen in der Registerdatei **124** für 28 Einträge, die vorher Mikrobefehlen des aktiven Threads zugeordnet worden sind, die Zuordnungen aufgehoben. Der Inhalt der Register mit aufgehobener Zuordnung für den aktiven Thread wird in einem schnellen Hilfsspeicher gespeichert, der eine Registeranordnung oder einen Direktzugriffsspeicher (RAM) umfassen kann, der mit einem Steuerregisterbus in dem multithreading-fähigen Prozessor **30** gekoppelt ist.

[0140] Die Aufhebung der Zuordnung der Registereinträge in der Registerdatei **124** kann durch eine Zuordnungsaufhebungs-Mikrocodesequenz ausgeführt werden, die von dem Mikrocode-Sequencer **66** als Reaktion auf ein Ereignis STOPCLK, HALT (HLT) oder SHUTDOWN fü den aktiven Thread ausgegeben wird. Die Zuordnungsaufhebungs-Mikrocodesequenz entfernt Aufzeichnungen bzw. Datensätze für die Registerdateieinträge in dem freien Listenmanager **134** (oder macht diese ungültig), und um Aufzeichnungen bzw. Datensätze für die Registerdateieinträge in der Trash-Heap-Array **132** zu erzeugen (oder zu validieren). Anders ausgedrückt werden Datensätze für die Zuordnungsaufhebungs-Registerdateieinträge von dem freien Listenmanager **134** zu der Trash-Heap-Array **132** durch die Mikrocodesequenz mit aufgehobener Zuordnung übertragen.

[0141] Die Abbildung aus Fig. 16B zeigt eine schematische Darstellung eines exemplarischen Ausführungsbeispiels der Operationen, die in dem Block **602** ausgeführt werden können. Zum Beispiel wird gemäß der Abbildung die Übertragung des Inhalts einer ersten Gruppe von Registern in der Registerdatei **124**, die vorher einem ersten Thread (z. B. Thread T0) zugeordnet waren, in den schnellen Hilfsspeicher (Scratch Pad) ausgeführt. Zu den zusätzlichen Operationen, die beim Speichern bzw. Sichern des Zustands ausgeführt werden können, zählt das Speichern der Inhalte der Architekturregister für einen austretenden Thread zu dem Hilfsspeicher sowie auch die Speicherung der Inhalte temporärer Mikrocode-Register, die dem ersten Thread zugeordnet sind, an den schnellen Hilfsspeicher beim Verlassen in diesem ersten Thread. Die beim Verlassen eines Threads frei gelassenen Register stehen danach für die neue Zuordnung zu einem anderen Thread (z. B. T1) zur Verfügung.

[0142] Nach dem erneuten Eintritt in einem bestimmten Thread (z. B. T0) wird hiermit festgestellt, dass der Inhalt der Register, die diesem Thread zugeordnet sind, wiederhergestellt werden kann aus dem schnellen Hilfsspeicher, wie dies in der Abbildung aus Fig. 16B durch die gestrichelte Linie dargestellt ist.

[0143] In dem Block **604** wird ein Thread-spezifischer „Eingrenzungsmikrobefehl" für den austretenden Thread in den Mikrobefehlsfluss für den austretenden Thread eingeführt, um alle etwaigen verbliebenen anstehenden Speicherzugriffe aus dem Speicheranordnungspuffer **48**, verschiedenen Cache-Speichern und den Prozessorbussen zu leeren bzw. löschen, die dem Thread zugeordnet sind. Diese Operation wird nicht zurückgezogen, bis nicht alle diese Blöcke abgeschlossen sind.

[0144] Da diese Ausführungseinheiten **20** Mikrobefehle verhältnismäßig schnell ausführen, werden alle neu dem Eingang der Ausführungseinheit hinzugefügte Mikrobefehle mit der Aktivierung des Nuke-Signals als Reaktion auf das Erkennen des Ereignisses SLEEP gelöscht. Wie dies bereits vorstehend im Text beschrieben worden ist, wird das Nuke-Signal **170** über einen ausreichenden Zeitraum (z. B. drei Taktzyklen) gehalten, um es zu ermöglichen, dass Mikrobefehle, die vor der Aktivierung des Nuke-Signals **170** in die Ausführungseinheit **70** eingetreten sind, aus dieser austreten. Wenn diese Mikrobefehle aus der Ausführungseinheit **70** austreten, werden sie gelöscht und die Rückschreibungen werden aufgehoben.

[0145] In dem Block **606** wird das Register „Unwind" **208** (Vorsetzen), das in dem Ereignisdetektor **188** verwaltet wird, durch einen Mikrobefehl so gesetzt, dass es anzeigt, dass sich der austretende Thread in einem inaktiven (oder Ruhezustand) Zustand befindet, wobei der durch den Mikrocode-Sequencer **66** erzeugte Befehl einen Wert zurückschreibt, der den Zustand des Registers Unwind festlegt.

[0146] In dem Block **608** werden die Ereignissperrregister **206** für den austretenden Thread gesetzt, um nicht unterbrechende Ereignisse für den austreten-

den Thread durch Schreib-Mikrobefehle des Steuerregisters, die von dem Mikrocode-Sequencer **666** ausgegeben werden, zu sperren. Das Festlegen bzw. Einstellen des Ereignissperrregisters für den austretenden Thread, angewiesen durch den Steuerregister-Mikrobefehl, ist abhängig von der Art bzw. dem Typ des behandelten Ruhezustandsereignisses. Wie dies bereits vorstehend im Text beschrieben worden ist, eignen sich abhängig von dem Ereignis SLEEP, das den Übergang in die inaktive Stufe ausgelöst hat, nur bestimmte Ereignisse als Unterbrechungsereignisse in Bezug auf den inaktiven Thread. Die Bestimmung, ob sich ein Ereignis als ein Unterbrechungsereignis für einen bestimmten inaktiven Thread eignet, erfolgt in speziellem Bezug auf den Zustand des Ereignissperrregisters **206** für den inaktiven Thread.

[0147] In dem Block **612** wird das Ruhezustandsereignis für den austretenden Thread unter Verwendung eines besonderen Mikrobefehls angezeigt, der eine Ruhezustandscodierung in dem Writeback-Fehlerinformationsfeld des speziellen Mikrobefehls platziert.

[0148] Die Abbildung aus Fig. 17 zeigt ein Flussdiagramm, dass ein Verfahren **700** gemäß einem exemplarischen Ausführungsbeispiel zum Eintreten eines inaktiven Threads in einen aktiven Zustand beim Erkennen eines Ereignisses BREAK für den inaktiven Thread veranschaulicht. Das Verfahren **700** beginnt in dem Block **702** mit der Erkennung des Auftretens eines Ereignisses für ein Ereignis, das sich als Ereignis BREAK (Unterbrechungsereignis) in Bezug auf einen inaktiven Thread eignen oder nicht eignen kann. In dem Entscheidungskästchen **703** wird durch die Ereigniserkennungslogik **185** für das relevante Ereignis bestimmt, ob sich das Ereignis als ein Ereignis BREAK für den inaktiven Thread eignet. Zu diesem Zweck prüft die Ereigniserkennungslogik **185** die Ereignissperrregister **206** in den Registern **200** des Ereignisdetektors **188**. Wenn der relevante Ereignistyp nicht als ein gesperrtes Ereignis BREAK angezeigt wird in Bezug auf den inaktiven Thread, so fährt das Verfahren **700** mit dem Block **704** fort, in dem die Takte nach Bedarf eingeschaltet werden, wobei das Ereignis normal signalisiert wird (auf einen Nuke-fähigen bzw. auslösbaren Punkt in dem anderen Thread wartend), und wobei die Behandlungsroutine wie für jedes andere Ereignis aufgerufen wird. Die Ereignisbehandlungsroutine prüft den Ruhezustand des Threads und wenn dieser gesetzt ist, fährt er in dem Block **706** mit der Wiederherstellung des Mikrocode-Zustands fort. Die Ereignisbehandlungsroutine **67** bestätigt den inaktiven Zustand des Threads durch den Zugriff auf das Unwind-Register **208**.

[0149] Im Besonderen fährt die Ereignisbehandlungsroutine **67** mit der Wiederherstellung des Mikrocode-Zustands für den eintretenden Thread fort, indem der vollständige gespeicherte Registerzustand, der Zustand des Sperrregisters und die Befehlszeigerinformationen wiederhergestellt werden.

[0150] Nach der Wiederherstellung des Mikrocode-Zustands in dem Block **706** fährt das Verfahren **700** mit dem Block **708** fort, in dem der Architekturzustand für den eintretenden Thread gespeichert wird. In dem Block **710** wird das Ereignissperrregister **206** für den eintretenden Thread durch einen entsprechenden Mikrobefehl zurückgesetzt oder gelöscht, der durch den Mikrocode-Sequencer **66** ausgegeben worden ist. In dem Block **712** fährt die Ereignisbehandlungsroutine **67** mit der Behandlung des Ereignisses BREAK fort. An diesem Punkt wird Mikrocode in dem multithreading-fähigen Prozessor **30** ausgeführt, wobei der Code die Ereignisbehandlungsroutine **67** bildet, um eine Rehe von Operationen auszuführen als Reaktion auf das Eintreten des Ereignisses. In dem Block **716** werden erneut Befehlserfassungsoperationen in dem Prozessor **30** für den eintretenden Thread aufgenommen. Das Verfahren **700** endet danach in dem Block **718**.

Taktsteuerlogik

[0151] Zur Reduzierung des Energieverbrauchs und der Wärmeausstrahlung in dem multithreading-fähigen Prozessor **30** ist es wünschenswert, mindestens einige Taktsignale in dem Prozessor **30** unter bestimmten Bedingungen anzuhalten bzw. zu stoppen oder anzuhalten. Die Abbildung aus Fig. 18 zeigt ein Flussdiagramm, das ein Verfahren **800** gemäß einem exemplarischen Ausführungsbeispiel des Anhaltens oder Unterbrechens ausgewählter Taktsignale in einem multithreading-fähigen Prozessor veranschaulicht, wie zum Beispiel dem vorstehend im Text beschriebenen exemplarischen Prozessor **30**. Für die Zwecke der vorliegenden Patentschrift sollen Verweise auf die Unterbrechung oder das Anhalten von Taktsignalen in dem Prozessor eine Reihe von Techniken zum Unterbrechen oder Anhalten eines Taktsignals oder von Signalen in dem Prozessor **30** einschließen. Zum Beispiel kann ein Phasenregelkreis (PLL) in dem Prozessor **30** unterbrochen werden, oder es kann die Verteilung eines Kerntaktsignals entlang einem Taktrücken gesperrt werden, oder es kann die Verteilung eines Taktsignals über den Taktrücken zu einzelnen funktionalen Einheiten in dem Prozessor torgesteuert oder anderweitig verhindert werden. Ein Ausführungsbeispiel der vorliegenden Erfindung zieht die letztgenannte Situation in Betracht, wobei die Versorgung eines internen Taktsignals an funktionale Einheiten in dem Prozessor **30** unterbrochen oder angehalten wird, und zwar von funktionaler Einheit zu funktionaler Einheit. Somit kann das interne Taktsignal bestimmten funktionalen Einheiten zugeführt werden, während es in Bezug auf andere funktionale Einheiten torgesteuert wird. Eine derartige Anordnung wird in dem Kontext eines Mi-

kroprozessors mit einem einzigen Thread in dem U.S. Patent US-A-5.655.127 beschrieben.

[0152] Das in der Abbildung aus Fig. 18 veranschaulichte Verfahren **800** kann in einem Ausführungsbeispiel durch die Taktsteuerlogik **35** ausgeführt werden, die in der Busschnittstelleneinheit **32** des Prozessors **30** enthalten ist. In alternativen Ausführungsbeispielen kann sich die Taktsteuerlogik **35** natürlich auch an anderen Stellen als an dem Prozessor **30** befinden. Die Abbildungen der Fig. 19A und Fig. 19B zeigen entsprechend ein Blockdiagramm bzw. ein schematisches Diagramm, welche weitere Einzelheiten in Bezug auf die exemplarische Taktsteuerlogik **35** veranschaulichen.

[0153] In erstem Bezug auf die Abbildung aus Fig. 19A empfängt die Taktsteuerlogik **35** gemäß der Abbildung drei primäre Eingaben: (1) aktive Bits **820** (z. B. T0_ACTIVE und T1_ACTIVE) gemäß der Ausgabe über die aktive Thread-Zustandsmaschine **174**; (2) die Signale **211** für detektierte Ereignisse, ausgegeben durch den Ereignisdetektor **188**; und (3) ein Snoop-Steuersignal **822**, das von der Busschnittstelleneinheit **32** ausgegeben wird, welche einen snoopfähigen Zugriff auf den Bus detektiert und das Signal **882** aktiviert. Die Taktsteuerlogik **35** verwendet diese Eingaben zum Erzeugen eines Stopptaktsignals **826**, das nacheinander die Taktung bestimmter funktionaler Einheiten in dem Prozessor **30** unterdrückt oder sperrt.

[0154] Die Abbildung aus Fig. 19B zeigt eine Prinzipskizze einer exemplarischen kombinatorischen Logik, welche die Eingaben **211**, **820** und **822** verwendet, um das Stopptaktsignal **826** auszugeben. Im Besonderen stellen die Ereignisdetektorsignale **211** eine Eingabe in ein ODER-Gatter **822** bereit, das wiederum eine Eingabe in ein weiteres ODER-Gatter **824** bereitstellt. Die aktiven Bits **820** und das Snoop-Steuersignal **822** stellen ebenfalls eine Eingabe in das NOR-Gatter **824** bereit, welches eine ODER-Funktion an diesen Eingaben ausführt, um das Stopptaktsignal **826** auszugeben.

[0155] In besonderem Bezug auf die Abbildung aus Fig. 18 beginnt das Verfahren **800** an dem Entscheidungskästchen **802** mit einer Bestimmung, ob etwaige Threads (z. B. ein erster und ein zweiter Thread) in dem multithreading-fähigen Prozessor **30** aktiv sind. Diese Bestimmung wird durch die Ausgabe der aktiven Bits **820** an das ODER-Gatter **824** in Fig. 19B reflektiert. Das exemplarische Ausführungsbeispiel veranschaulicht, dass die Bestimmung in Bezug auf die beiden Threads erfüllt werden kann, wobei hiermit festgestellt wird, dass diese Bestimmung in Bezug auf eine beliebige Anzahl von Threads vorgenommen wird, die in einem multithreading-fähigen Prozessor unterstützt werden.

[0156] Nach einer negativen Bestimmung in dem Entscheidungskästchen **802** fährt das Verfahren **800** mit dem Entscheidungskästchen **804** fort, wo bestimmt wird, ob eines der Ereignisse, die nicht gesperrt sind, für einen in dem multithreading-fähigen Prozessor unterstützten Thread anstehen. In dem exemplarischen Ausführungsbeispiel umfasst dies die Bestimmung, ob etwaige Ereignisse für einen ersten oder einen zweiten Thread anstehen. Diese Bestimmung ist dargestellt durch die Eingabe der Signale **211** für detektierte Ereignisse in das ODER-Gatter **822** aus der Abbildung aus Fig. 19B.

[0157] Nach einer negativen Bestimmung in dem Entscheidungskästchen **804** erfolgt eine weitere Bestimmung in dem Entscheidungskästchen **806**, ob etwaige Snoops (z. B. Bus-Snoops, SNC-Snoops oder sonstige Snoops) von dem Prozessorbus verarbeitet werden. In dem exemplarischen Ausführungsbeispiel der vorliegenden Erfindung wird diese Bestimmung implementiert durch die Eingabe des Snoop-Steuersignals **822** in das ODER-Gatter **824**.

[0158] Nach der negativen Bestimmung in dem Entscheidungskästchen **806** fährt das Verfahren **800** mit dem Block **808** fort, in dem interne Taktsignale an ausgewählte funktionale Einheiten angehalten oder unterdrückt werden. Im Besonderen werden die Taktsignale an die Busabfertigungslogik und die Buszugriffslogik nicht angehalten oder gestoppt, da es dies der Busschnittstelleneinheit **32** ermöglicht, Unterbrechungsereignisse (BREAK) oder an dem Systembus auftretende Snoops (z. B. Pin-Ereignisse) zu detektieren und die Takte an die funktionalen Einheiten neu zu beginnen als Reaktion auf derartige Unterbrechungsereignisse. Die Unterdrückung der internen Taktsignale an funktionale Einheiten wird implementiert durch die Aktivierung des Stopptaktsignals **826**, das den Effekt der Torsteuerung des Taktsignals zu vorbestimmten funktionalen Einheiten aufweist.

[0159] Nach der vollständigen Ausführung des Blocks **808** kehrt das Verfahren **800** zurück zu dem Entscheidungskästchen **802**. Nach den Bestimmungen an den Entscheidungskästchen **802**, **804** und **806** kann ein kontinuierlicher Schleifenbetrieb erfolgen.

[0160] Nach einer positiven Bestimmung in einem der Entscheidungskästchen **802**, **804** und **806** verzweigt das Verfahren **800** zu dem Block **810**, in dem wenn Taktsignale zu bestimmten funktionalen Einheiten torgesteuert werden, diese internen Taktsignale danach erneut aktiviert werden. Wenn die Taktsignale alternativ bereits aktiv sind, werden diese Taktsignale in einem aktiven Zustand gehalten.

[0161] Wenn der Block **810** als Reaktion auf ein Unterbrechungsereignis ausgeführt wird (z. B. nach einer positiven Bestimmung in dem Entscheidungs-

kästchen **804**), können die funktionalen Einheiten in dem Mikroprozessor auf die vorstehend beschriebene Art und Weise aktiv unterteilt werden, auf der Basis der Anzahl aktiver Threads bei Aktivierung des Nuke-Signals. In einem multithreading-fähigen Prozessor **30** mit zwei oder mehr Threads können zum Beispiel einige dieser Threads inaktiv sein, wobei die funktionalen Einheiten in diesem Fall nicht aufgeteilt werden, um die inaktiven Threads zu berücksichtigen.

[0162] Nach Abschluss des Blocks **810** spring das Verfahren **800** wiederum zurück zu dem Entscheidungskästchen **802** und beginnt eine weitere Iteration der Entscheidungen, die durch die Entscheidungskästchen **802**, **804** und **806** dargestellt sind.

[0163] Beschrieben wurden somit ein Verfahren und eine Vorrichtung zur Verwaltung eines Taktsignals in einem multithreading-fähigen Prozessor. Die vorliegende Erfindung wurde zwar in Bezug auf bestimmte exemplarische Ausführungsbeispiele beschrieben, wobei jedoch ersichtlich ist, dass an diesen Ausführungsbeispielen verschiedene Modifikationen und Abänderungen vorgenommen werden können, ohne dabei vom Umfang der vorliegenden Erfindung gemäß den Definitionen in den Ansprüchen abzuweichen.

Patentansprüche

1. Verfahren, das folgendes aufweist:
das Verwalten einer Anzeige eines anstehenden Ereignisses (**211**) in Bezug auf mehrere Threads, die in einem multithreading-fähigen Prozessor (**30**) unterstützt werden;
das Verwalten einer Anzeige eines aktiven oder inaktiven Zustands (**820**) für jeden der mehreren Threads, die in dem multithreading-fähigen Prozessor (**30**) unterstützt werden;
das Detektieren eines Taktdeaktivierungszustands (**826**), der durch die Anzeige von keinen anstehenden Ereignissen (**211**) in Bezug auf jeden der mehreren Threads angezeigt wird sowie einen inaktiven Zustand (**820**) für jeden der mehreren Threads; und
das Deaktivieren eines Taktsignals (**304**, **826**), sofern freigegeben, in Bezug auf mindestens eine funktionale Einheit in dem multithreading-fähigen Prozessor (**30**) als Reaktion auf das Detektieren des Taktdeaktivierungszustands (**826**).

2. Verfahren nach Anspruch 1, wobei das Verfahren das Detektieren eines Taktfreigabezustands aufweist, angezeigt durch die Anzeige eines anstehenden Ereignisses (**211**) in Bezug auf mindestens einen Thread der mehreren Threads, die in dem multithreading-fähigen Prozessor (**30**) unterstützt werden, oder durch die Anzeige eines aktiven Zustands (**820**) für mindestens einen Thread der mehreren Threads, die in dem multithreading-fähigen Prozessor (**30**) unterstützt werden, und das Freigeben des Taktsignals (**304**, **826**), sofern deaktiviert, in Bezug auf die mindestens eine funktionale Einheit in dem multithreading-fähigen Prozessor (**30**) als Reaktion auf das Detektieren des Taktfreigabezustands.

3. Verfahren nach Anspruch 2, wobei das Verfahren das Verwalten einer Ereignissperranzeige für mindestens einen ersten Thread der Mehrzahl von Threads aufweist, die von dem multithreading-fähigen Prozessor (**30**) unterstützt werden, wobei die Ereignissperranzeige mindestens ein Ereignis in Bezug auf den ersten Thread identifiziert, das kein Ereignis darstellt, das einen Übergang des ersten Threads zwischen den inaktiven und aktiven Zuständen (**820**) auslöst.

4. Verfahren nach Anspruch 3, wobei das Detektieren des Taktdeaktivierungszustands (**826**) und des Taktfreigabezustands der Ereignissperranzeige in Bezug auf den ersten Thread unterliegt, und wobei der Taktdeaktivierungszustand (**826**) detektiert wird oder ein Taktfreigabezustand nicht detektiert wird, wenn ein anstehendes Ereignis (**211**) durch die Ereignissperranzeige für den ersten Thread als gesperrt angezeigt wird.

5. Verfahren nach Anspruch 1, wobei die Verwaltung der Anzeige der Thread-Aktivität in Bezug auf jeden der Mehrzahl von Threads das Verwalten einer aktiven Thread-Zustandsmaschine (**174**) umfasst, welche ein Signal (**820**) für jeden Thread der Mehrzahl der unterstützten Threads ausgibt, um anzuzeigen, ob sich ein entsprechender Thread in dem aktiven oder in dem inaktiven Zustand (**820**) befindet.

6. Vorrichtung, die folgendes umfasst:
einen Indikator für ein anstehendes Ereignis, der eine Anzeige für ein anstehendes Ereignis (**211**) verwaltet in Bezug auf jeden der Mehrzahl von Threads, die in einem multithreading-fähigen Prozessor (**30**) unterstützt werden;
einen Indikator für einen aktiven Thread, der eine Anzeige für einen aktiven oder einen inaktiven Zustand (**820**) für jeden Thread der Mehrzahl von Threads verwaltet, die in dem multithreading-fähigen Prozessor (**30**) unterstützt werden; und
eine Taktsteuerlogik (**35**) zum Detektieren eines Taktdeaktivierungszustands (**826**), der durch die Anzeige von keinen anstehenden Ereignissen (**211**) in Bezug auf jeden Thread der Mehrzahl von Threads angezeigt wird, und eines inaktiven Zustands (**820**) in Bezug auf jeden Thread der Mehrzahl von Threads, und zum Deaktivieren eines Taktsignals (**304**, **826**) in Bezug auf mindestens eine funktionale Einheit in dem multithreading-fähigen Prozessor (**30**) als Reaktion auf das Detektieren des Taktdeaktivierungszustands (**826**).

7. Vorrichtung nach Anspruch 6, mit einem Ereig-

nissperrindikator, der eine Ereignissperranzeige verwaltet für mindestens einen ersten Thread der Mehrzahl von Threads, die durch den multithreading-fähigen Prozessor (**30**) unterstützt werden, wobei die Ereignissperranzeige mindestens ein Ereignis in Bezug auf den ersten Thread identifiziert, das kein Ereignis darstellt, das einen Übergang des ersten Threads zwischen den inaktiven und aktiven Zuständen (**820**) auslöst.

8. Vorrichtung nach Anspruch 6, wobei die Vorrichtung eine Bus-Snoop-Logik aufweist, die eine Anzeige eines Buszugriffszustands (**822**) bereitstellt, und wobei die Taktsteuerlogik (**35**) den Taktdeaktivierungszustand (**826**) nur in Abwesenheit einer Anzeige einer Buszugriffsoperation von der Bus-Snoop-Logik detektiert.

9. Vorrichtung nach Anspruch 8, mit einer aktiven Thread-Zustandsmaschine (**174**), die ein Signal (**820**) für jeden Thread der Mehrzahl von unterstützten Threads ausgibt, um anzuzeigen, ob sich ein entsprechender Thread in dem aktiven Zustand oder in dem inaktiven Zustand (**820**) befindet.

10. Vorrichtung nach Anspruch 6, wobei die Taktsteuerlogik (**35**) das Taktsignal (**304**, **826**) deaktiviert in Bezug auf die mindestens eine funktionale Einheit, wobei sie jedoch die Bereitstellung eines Taktsignals (**304**, **826**) an die Busstifte des multithreading-fähigen Prozessors (**30**) aufrechterhält, um das Detektieren einer Buszugriffsoperation oder eines Ereignisses zu ermöglichen, wenn das Taktsignal (**304**, **826**) deaktiviert wird.

Es folgen 23 Blatt Zeichnungen

Anhängende Zeichnungen

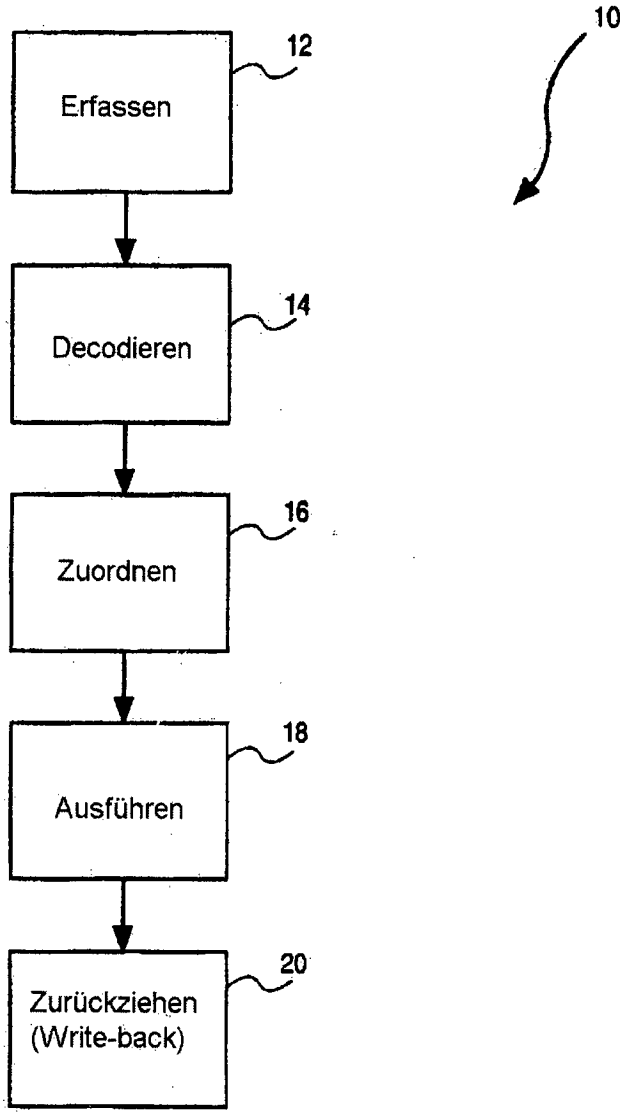


FIG. 1

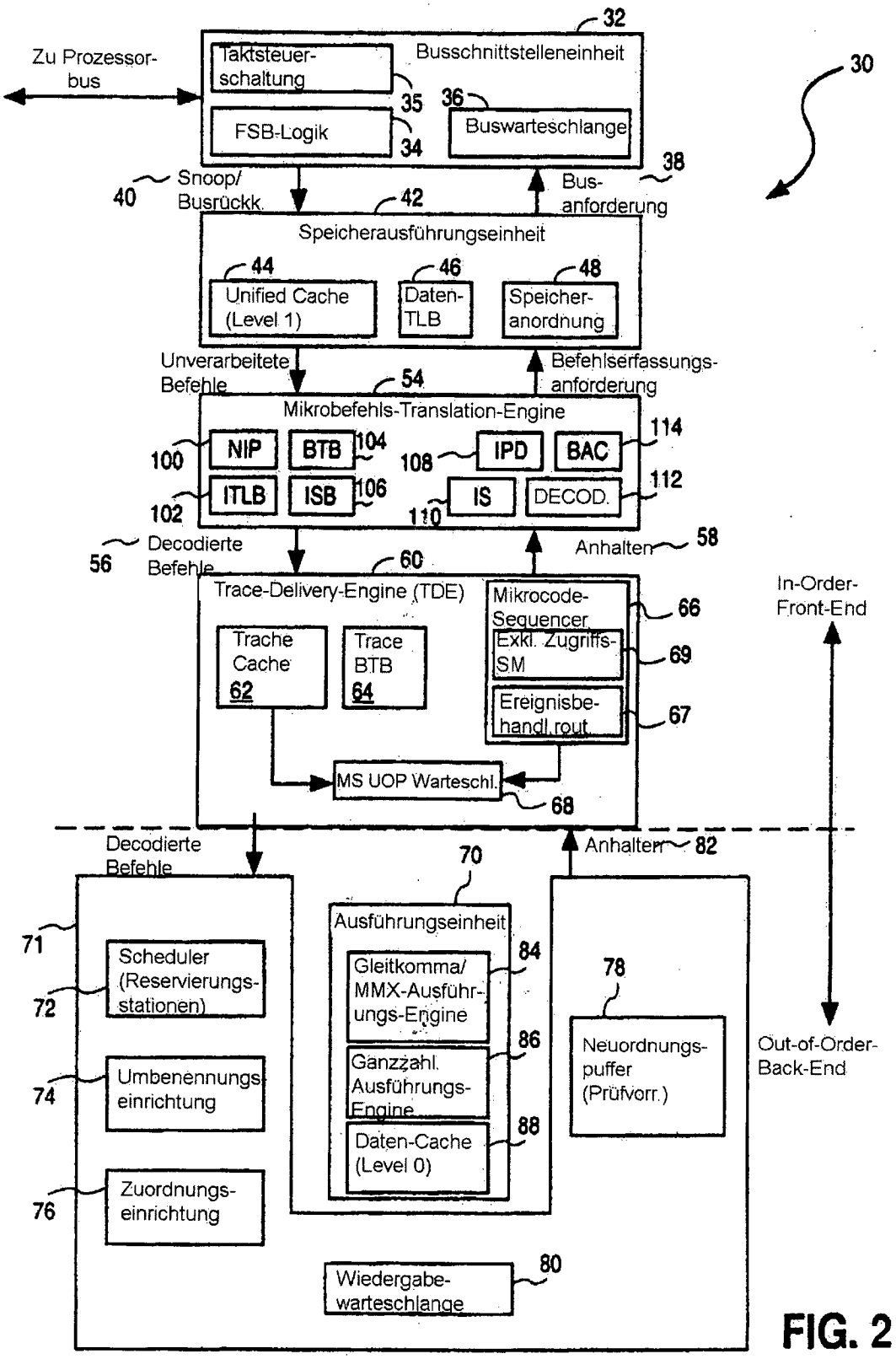

Zu Prozessor-bus
Taktsteuer-schaltung
Busschnittstelleneinheit
FSB-Logik
Buswarteschlange
Snoop/ Busrückk.
Bus-anforderung
Speicherausführungseinheit
Unified Cache (Level 1)
Daten-TLB
Speicher-anordnung
Unverarbeitete Befehle
Befehlserfassungs-anforderung
Mikrobefehls-Translation-Engine
NIP
BTB
IPD
BAC
ITLB
ISB
IS
DECOD.
Decodierte Befehle
Anhalten
Trace-Delivery-Engine (TDE)
Trache Cache 62
Trace BTB 64
Mikrocode-Sequencer
Exkl. Zugriffs-SM
Ereignisbe-handl.rout.
MS UOP Warteschl.
In-Order-Front-End
Decodierte Befehle
Anhalten
Ausführungseinheit
Scheduler (Reservierungs-stationen)
Umbenennungs-einrichtung
Zuordnungs-einrichtung
Gleitkomma/ MMX-Ausführ-ungs-Engine
Ganzzahl. Ausführungs-Engine
Daten-Cache (Level 0)
Neuordnungs-puffer (Prüfvorr.)
Wiedergabe-warteschlange
Out-of-Order-Back-End

FIG. 2

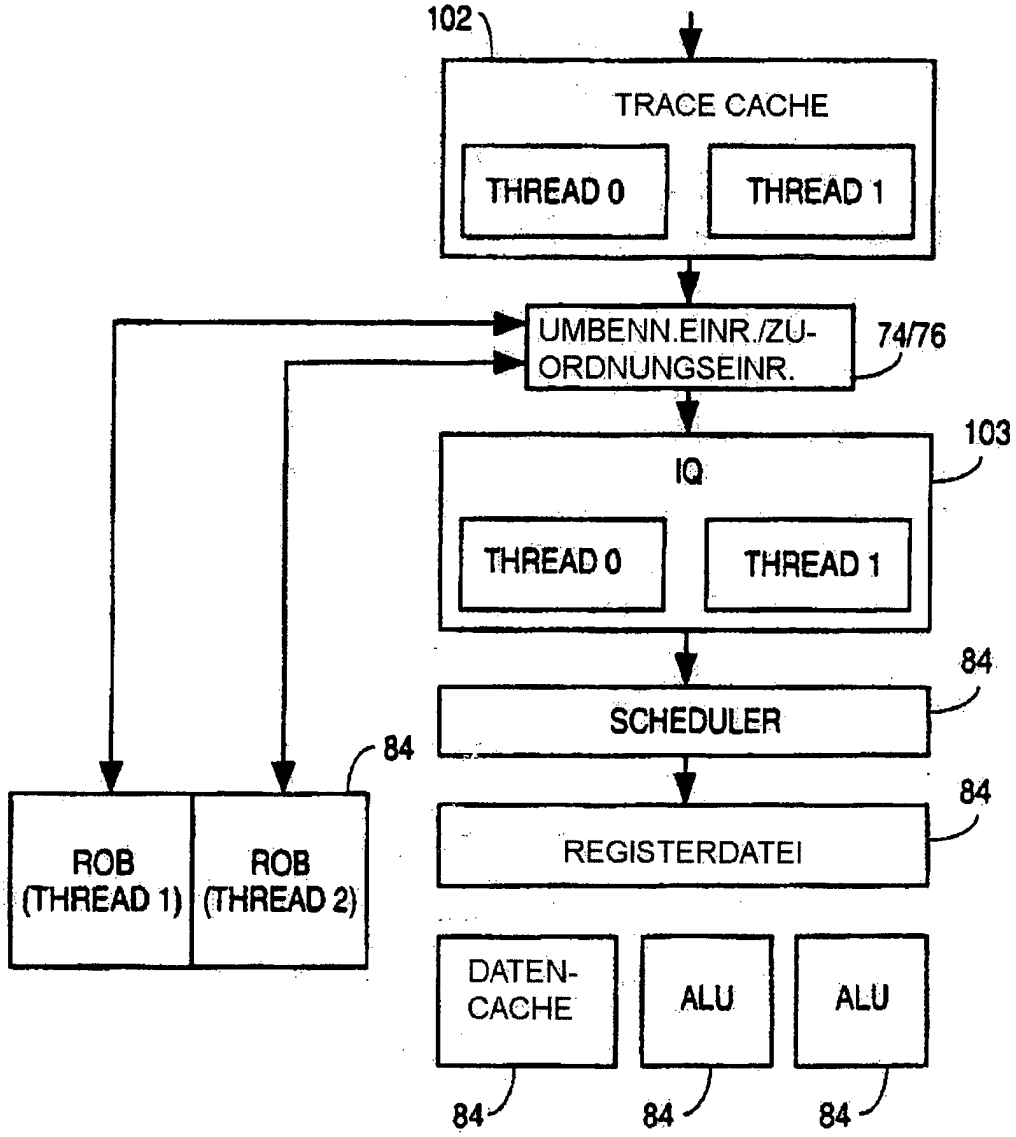

102
TRACE CACHE
THREAD 0
THREAD 1
UMBENN.EINR./ZU-ORDNUNGSEINR.
74/76
103
IQ
THREAD 0
THREAD 1
84
SCHEDULER
84
REGISTERDATEI
84
ROB (THREAD 1)
ROB (THREAD 2)
DATEN-CACHE
ALU
ALU
84
84
84

FIG. 3

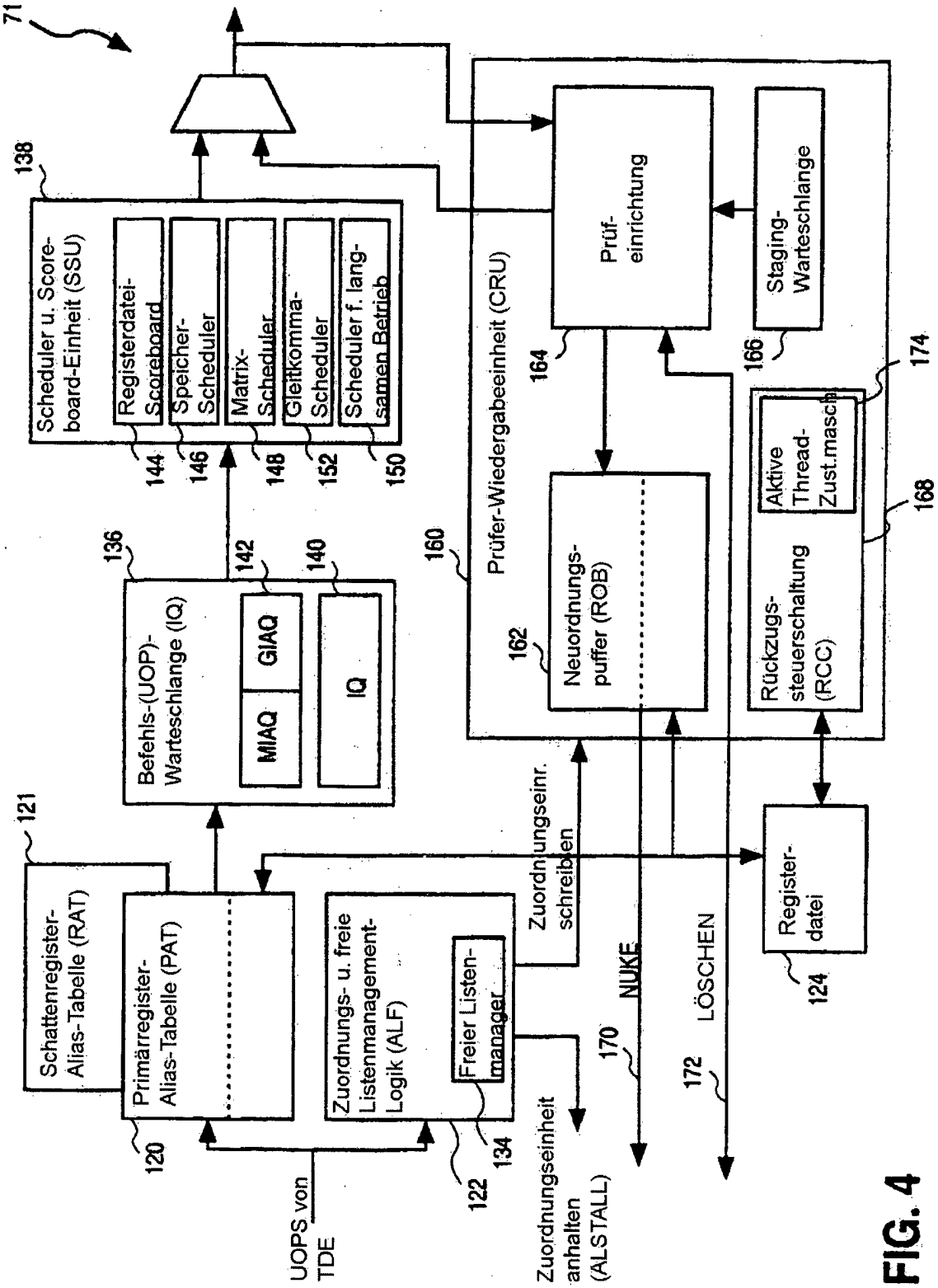

71
138
Scheduler u. Score-board-Einheit (SSU)
Registerdatei-Scoreboard
Speicher-Scheduler
Matrix-Scheduler
Gleitkomma-Scheduler
Scheduler f. lang-samen Betrieb
144
146
148
152
150
136
Befehls-(UOP)-Warteschlange (IQ)
MIAQ
GIAQ
IQ
142
140
160
Prüfer-Wiedergabeeinheit (CRU)
Neuordnungs-puffer (ROB)
162
Prüf-einrichtung
164
Staging-Warteschlange
166
Aktive Thread-Zust.masch
174
168
Rückzugs-steuerschaltung (RCC)
121
Schattenregister-Alias-Tabelle (RAT)
Primärregister-Alias-Tabelle (PAT)
120
Zuordnungs- u. freie Listenmanagement-Logik (ALF)
Freier Listen-manager
134
122
Zuordnungseinr. schreiben
Register-datei
124
NUKE
170
LÖSCHEN
172
UOPS von TDE
Zuordnungseinheit anhalten (ALSTALL)

FIG. 4

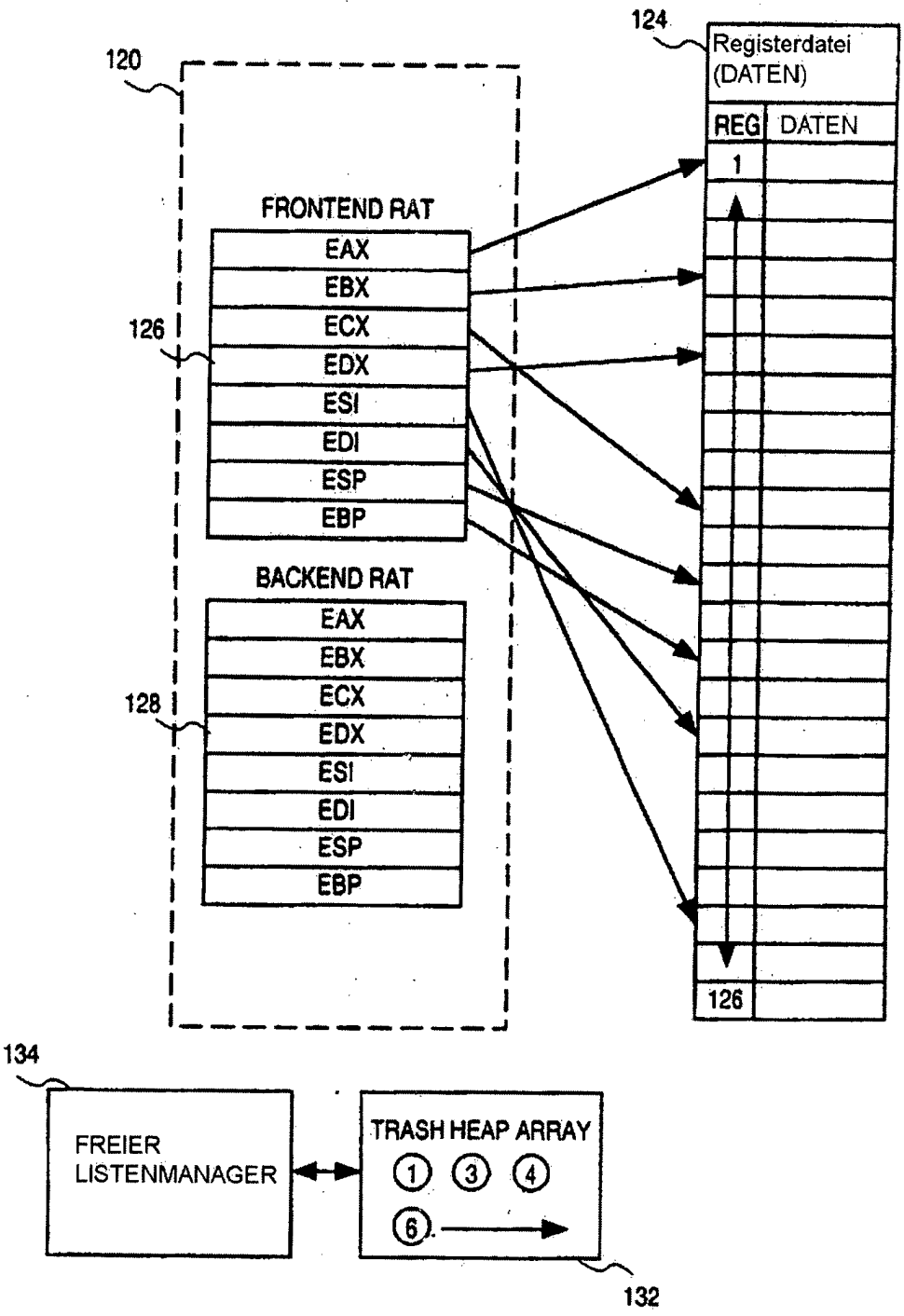
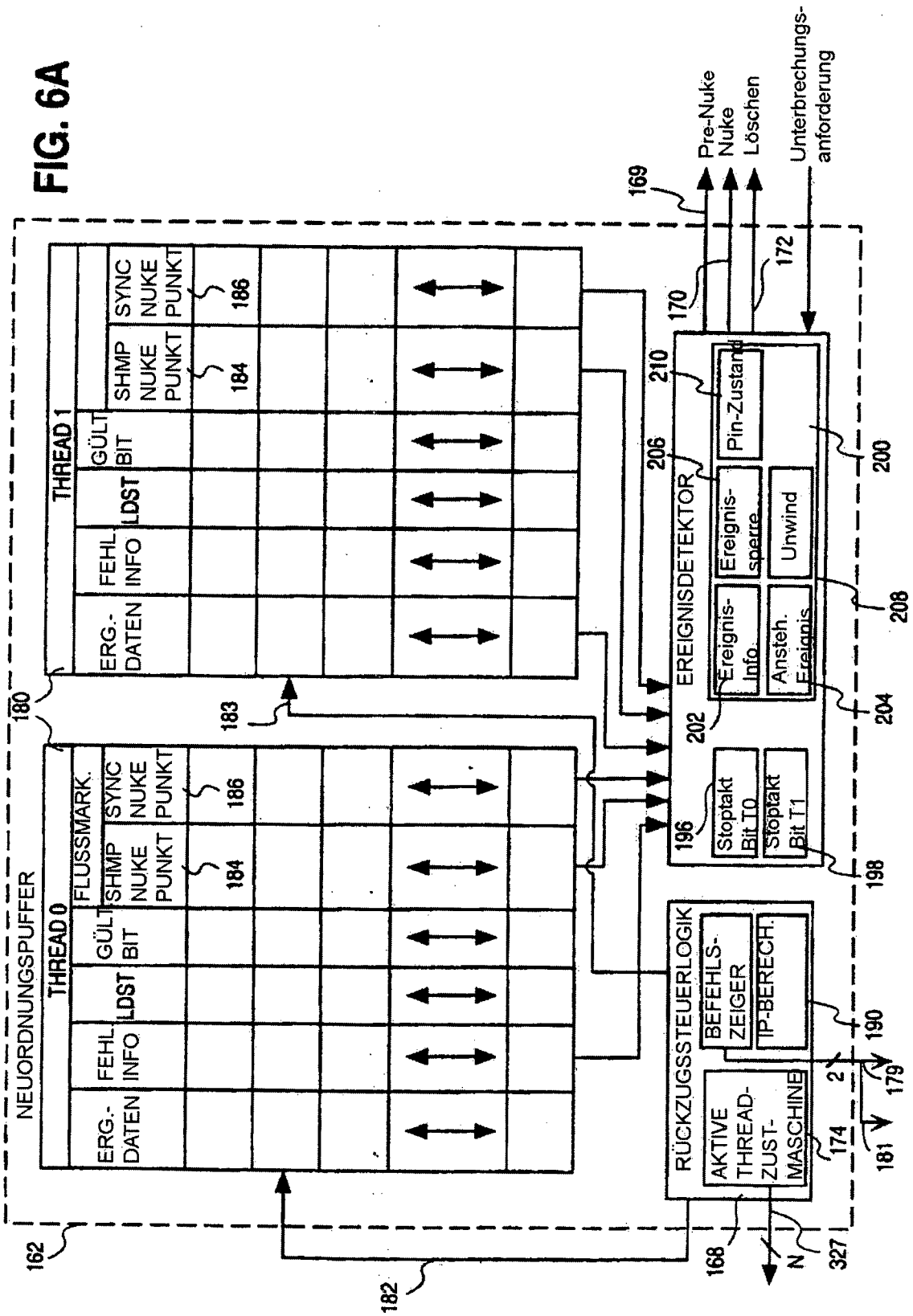

120
124
Registerdatei
(DATEN)
REG
DATEN
1
FRONTEND RAT
EAX
EBX
ECX
EDX
ESI
EDI
ESP
EBP
126
BACKEND RAT
EAX
EBX
ECX
EDX
ESI
EDI
ESP
EBP
128
126
134
FREIER
LISTENMANAGER
TRASH HEAP ARRAY
1
3
4
6
132

FIG. 5

FIG. 6A


NEUORDNUNGSPUFFER
THREAD 0
ERG.-DATEN
FEHL INFO
LDST
GÜLT BIT
SHMP NUKE PUNKT
FLUSSMARK.
SYNC NUKE PUNKT
THREAD 1
ERG.-DATEN
FEHL INFO
LDST
GÜLT BIT
SHMP NUKE PUNKT
SYNC NUKE PUNKT
EREIGNISDETEKTOR
Pin-Zustand
Ereignis-sperre
Unwind
Ereignis-Info.
Ansteh. Ereignis
Stoptakt Bit T0
Stoptakt Bit T1
RÜCKZUGSSTEUERLOGIK
BEFEHLS-ZEIGER
IP-BERECH.
AKTIVE THREAD-ZUST-MASCHINE
Pre-Nuke
Nuke
Löschen
Unterbrechungs-anforderung
162
168
169
170
172
174
179
180
181
182
183
184
186
190
196
198
200
202
204
206
208
210
327
N

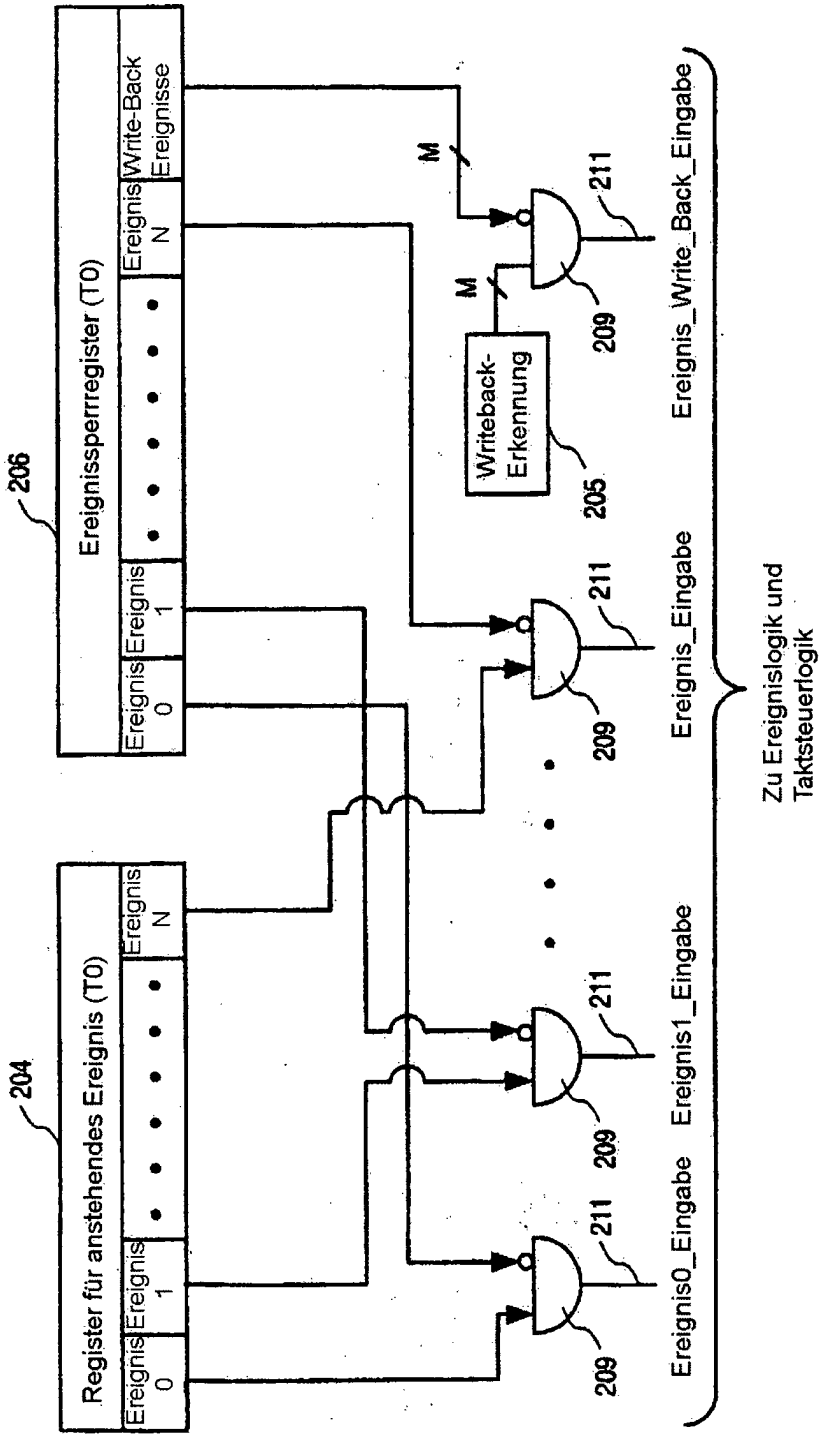

204
Register für anstehendes Ereignis (T0)
Ereignis 0
Ereignis 1
Ereignis N
206
Ereignissperrregister (T0)
Ereignis 0
Ereignis 1
Ereignis N
Write-Back Ereignisse
M
M
Writeback-Erkennung
205
209
211
Ereignis0_Eingabe
Ereignis1_Eingabe
Ereignis_Eingabe
Ereignis_Write_Back_Eingabe
Zu Ereignislogik und Taktsteuerlogik

FIG. 6B

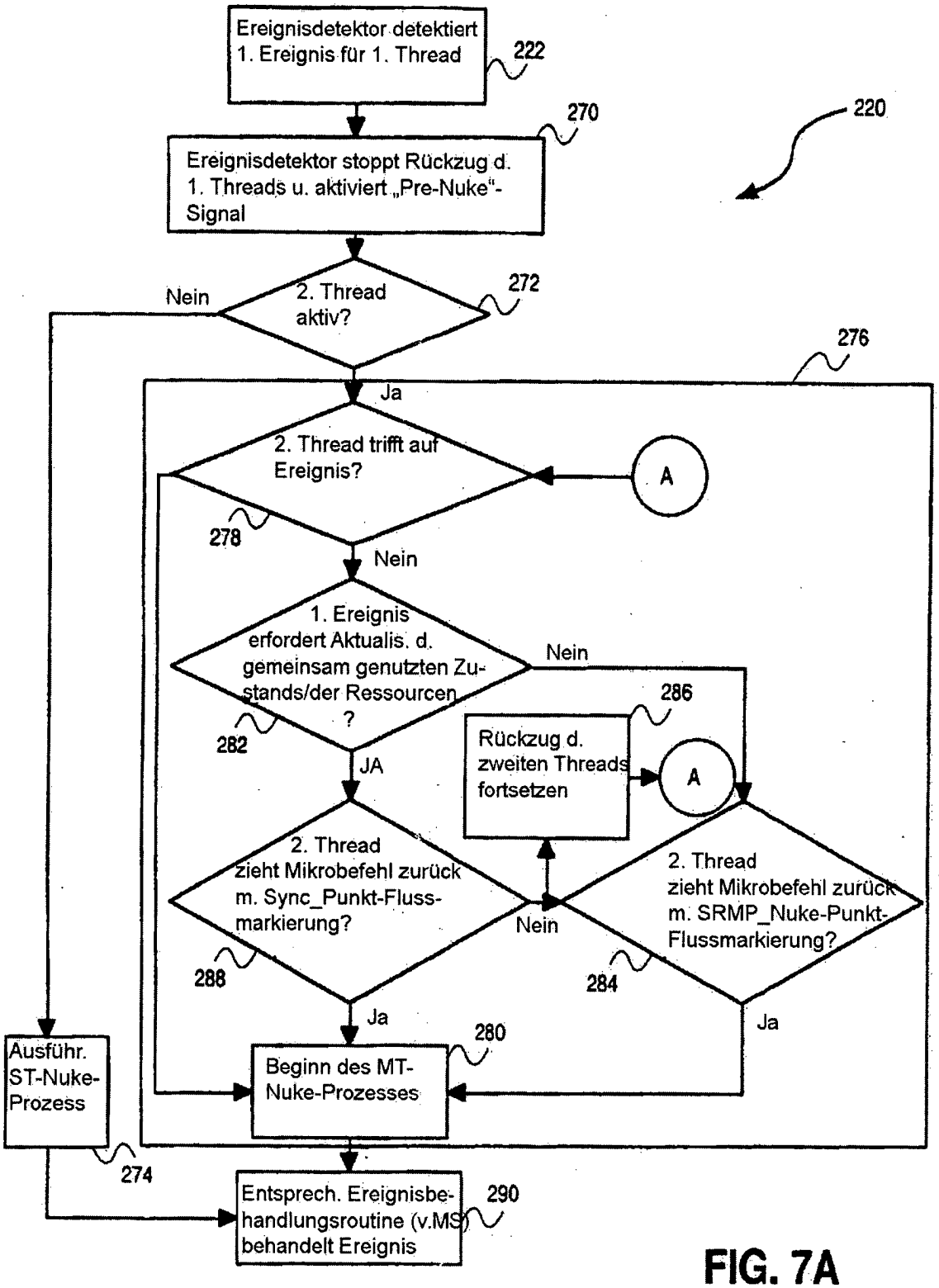

Ereignisdetektor detektiert 1. Ereignis für 1. Thread
222
220
270
Ereignisdetektor stoppt Rückzug d. 1. Threads u. aktiviert „Pre-Nuke"-Signal
Nein
2. Thread aktiv?
272
276
Ja
2. Thread trifft auf Ereignis?
A
278
Nein
1. Ereignis erfordert Aktualis. d. gemeinsam genutzten Zustands/der Ressourcen ?
Nein
286
282
JA
Rückzug d. zweiten Threads fortsetzen
A
2. Thread zieht Mikrobefehl zurück m. Sync_Punkt-Flussmarkierung?
Nein
2. Thread zieht Mikrobefehl zurück m. SRMP_Nuke-Punkt-Flussmarkierung?
288
284
Ja
Ja
280
Ausführ. ST-Nuke-Prozess
Beginn des MT-Nuke-Prozesses
274
Entsprech. Ereignisbehandlungsroutine (v.MS) behandelt Ereignis
290

FIG. 7A

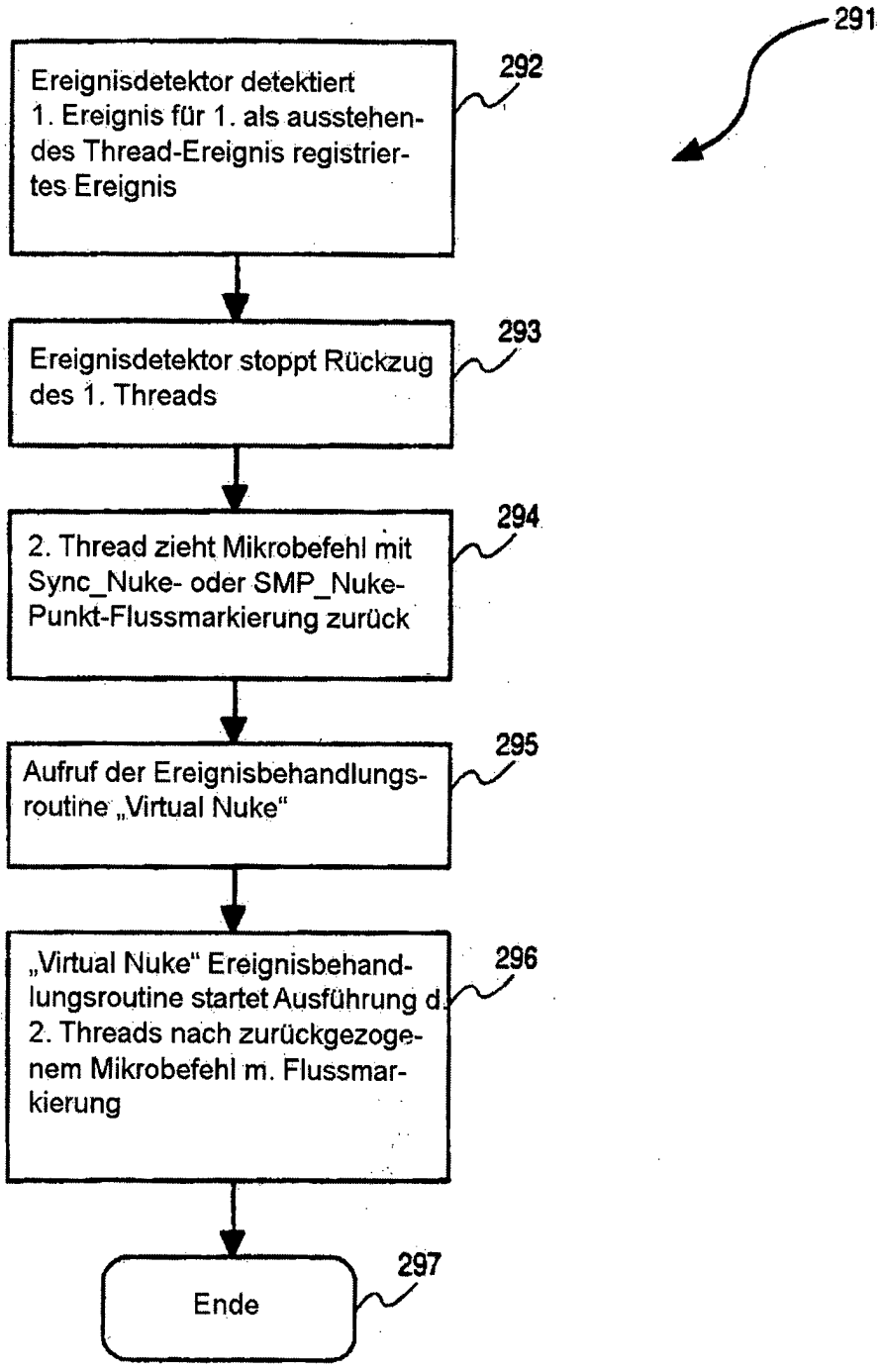

291
Ereignisdetektor detektiert 1. Ereignis für 1. als ausstehendes Thread-Ereignis registriertes Ereignis
292
Ereignisdetektor stoppt Rückzug des 1. Threads
293
2. Thread zieht Mikrobefehl mit Sync_Nuke- oder SMP_Nuke-Punkt-Flussmarkierung zurück
294
Aufruf der Ereignisbehandlungsroutine „Virtual Nuke"
295
„Virtual Nuke" Ereignisbehandlungsroutine startet Ausführung d. 2. Threads nach zurückgezogenem Mikrobefehl m. Flussmarkierung
296
Ende
297

FIG. 7B

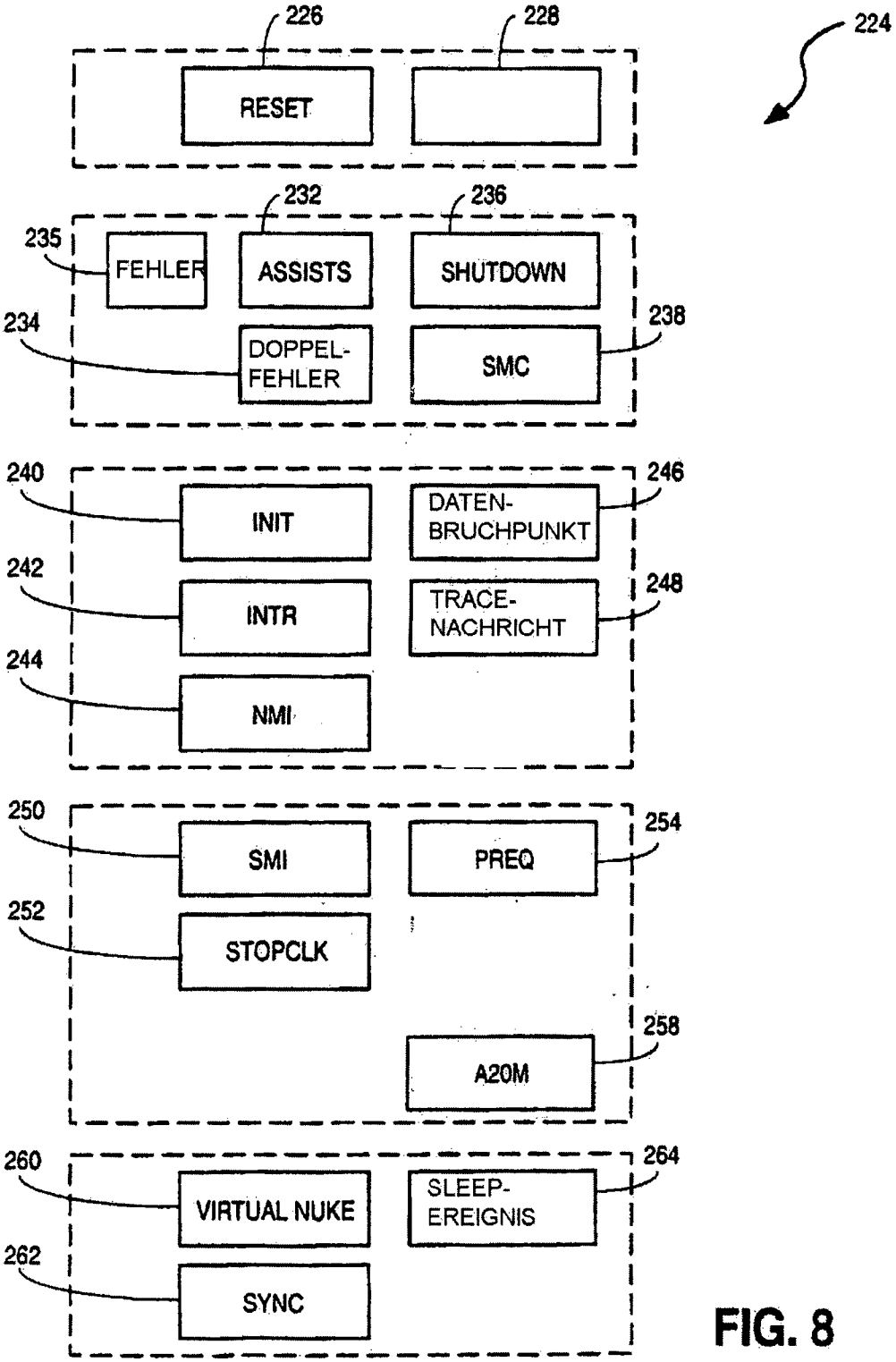

224
226
228
RESET
232
236
235
FEHLER
ASSISTS
SHUTDOWN
234
238
DOPPEL-
FEHLER
SMC
240
246
INIT
DATEN-
BRUCHPUNKT
242
248
INTR
TRACE-
NACHRICHT
244
NMI
250
254
SMI
PREQ
252
STOPCLK
258
A20M
260
264
VIRTUAL NUKE
SLEEP-
EREIGNIS
262
SYNC

FIG. 8

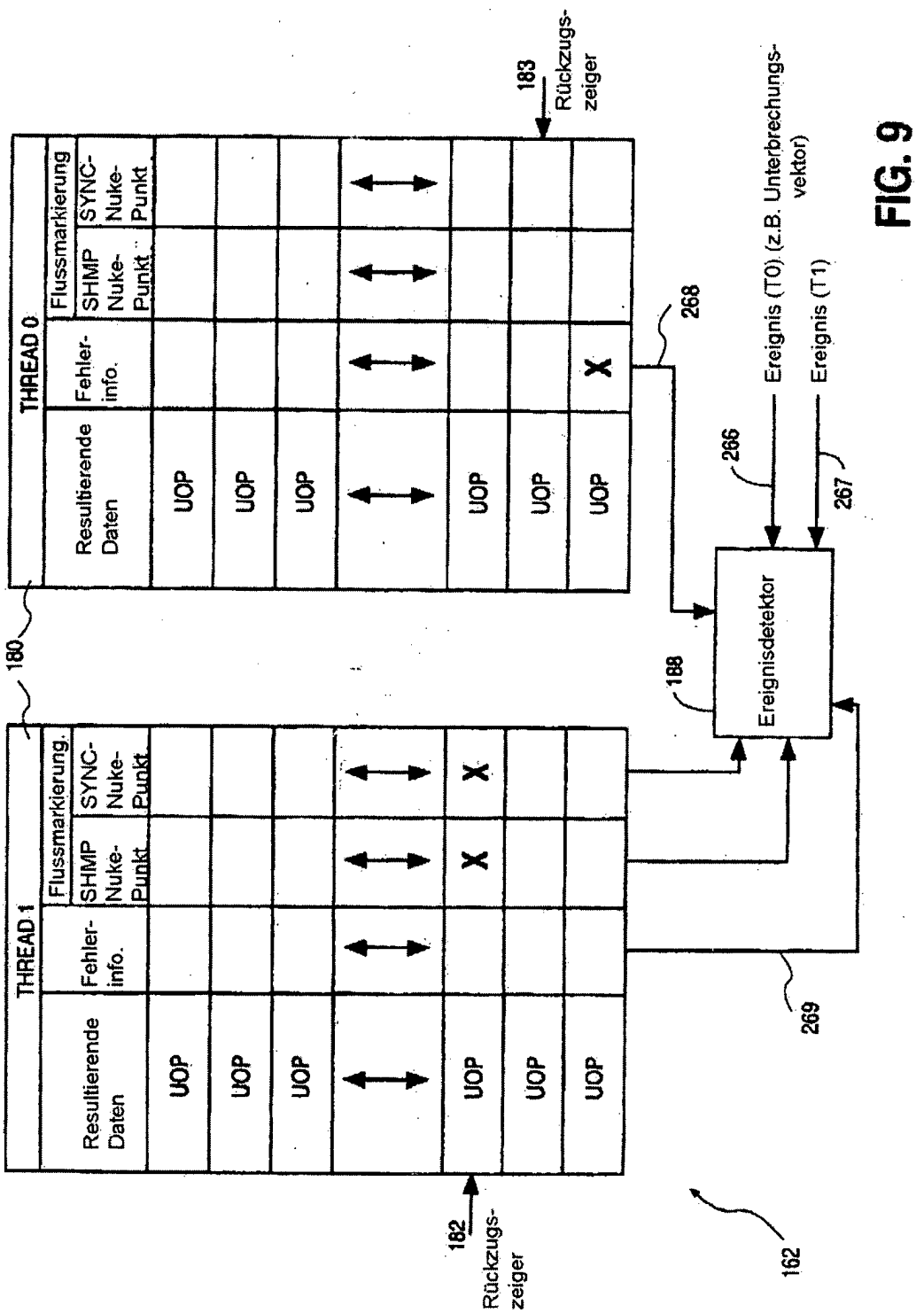

THREAD 0
Resultierende Daten
Fehler-info.
Flussmarkierung
SHMP Nuke-Punkt
SYNC-Nuke-Punkt
UOP
183
Rückzugs-zeiger
180
THREAD 1
Resultierende Daten
Fehler-info.
Flussmarkierung
SHMP Nuke-Punkt
SYNC-Nuke-Punkt.
UOP
X
182
Rückzugs-zeiger
162
268
266
267
269
188
Ereignisdetektor
Ereignis (T0) (z.B. Unterbrechungs-vektor)
Ereignis (T1)

FIG. 9

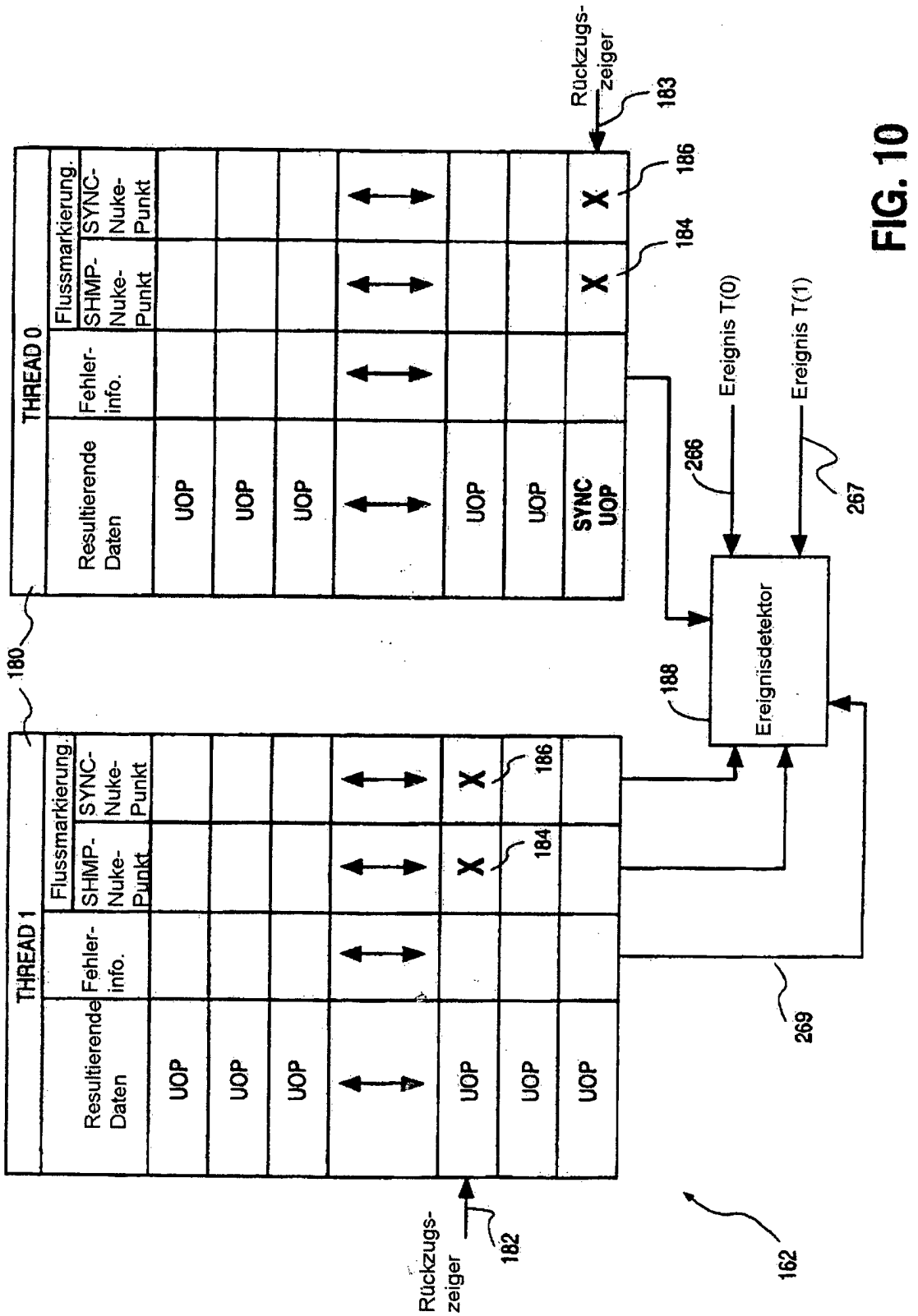

THREAD 0
Resultierende Daten
Fehler-info.
Flussmarkierung.
SHMP-Nuke-Punkt
SYNC-Nuke-Punkt
UOP
SYNC UOP
Rückzugs-zeiger
183
186
184
180
266
267
Ereignis T(0)
Ereignis T(1)
Ereignisdetektor
188
269
THREAD 1
182
162

FIG. 10

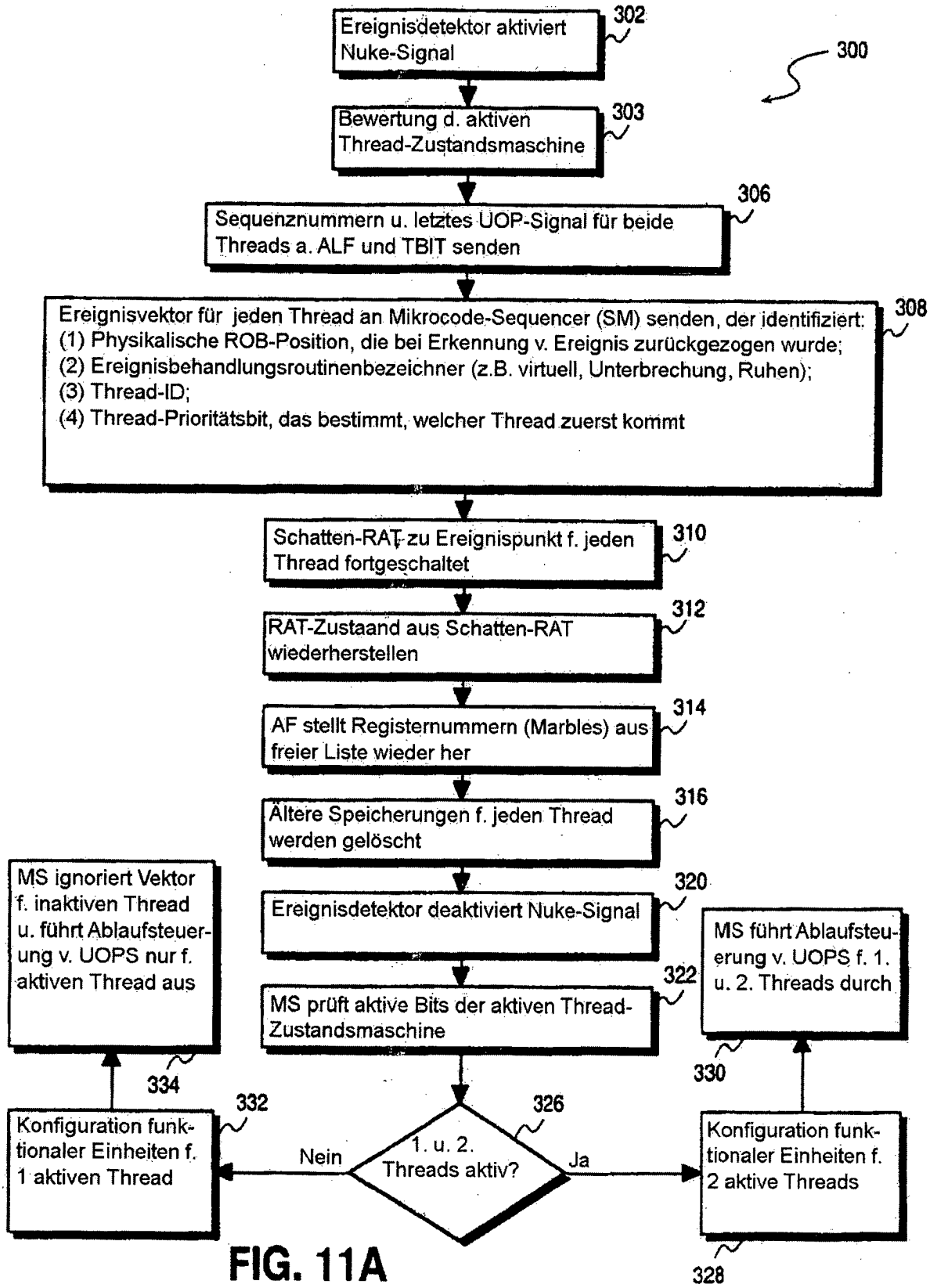

Ereignisdetektor aktiviert Nuke-Signal
302
300
Bewertung d. aktiven Thread-Zustandsmaschine
303
Sequenznummern u. letztes UOP-Signal für beide Threads a. ALF und TBIT senden
306
Ereignisvektor für jeden Thread an Mikrocode-Sequencer (SM) senden, der identifiziert:
(1) Physikalische ROB-Position, die bei Erkennung v. Ereignis zurückgezogen wurde;
(2) Ereignisbehandlungsroutinenbezeichner (z.B. virtuell, Unterbrechung, Ruhen);
(3) Thread-ID;
(4) Thread-Prioritätsbit, das bestimmt, welcher Thread zuerst kommt
308
Schatten-RAT zu Ereignispunkt f. jeden Thread fortgeschaltet
310
RAT-Zustaand aus Schatten-RAT wiederherstellen
312
AF stellt Registernummern (Marbles) aus freier Liste wieder her
314
Ältere Speicherungen f. jeden Thread werden gelöscht
316
Ereignisdetektor deaktiviert Nuke-Signal
320
MS prüft aktive Bits der aktiven Thread-Zustandsmaschine
322
1. u. 2. Threads aktiv?
326
Nein
Ja
MS ignoriert Vektor f. inaktiven Thread u. führt Ablaufsteuerung v. UOPS nur f. aktiven Thread aus
334
Konfiguration funktionaler Einheiten f. 1 aktiven Thread
332
MS führt Ablaufsteuerung v. UOPS f. 1. u. 2. Threads durch
330
Konfiguration funktionaler Einheiten f. 2 aktive Threads
328

FIG. 11A

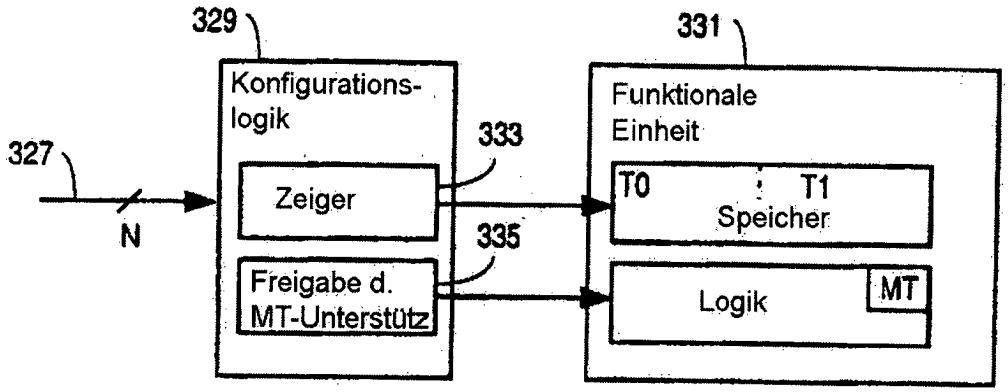

329
Konfigurations-
logik
327
N
Zeiger
Freigabe d.
MT-Unterstütz
333
335
331
Funktionale
Einheit
T0
T1
Speicher
Logik
MT

FIG. 11B

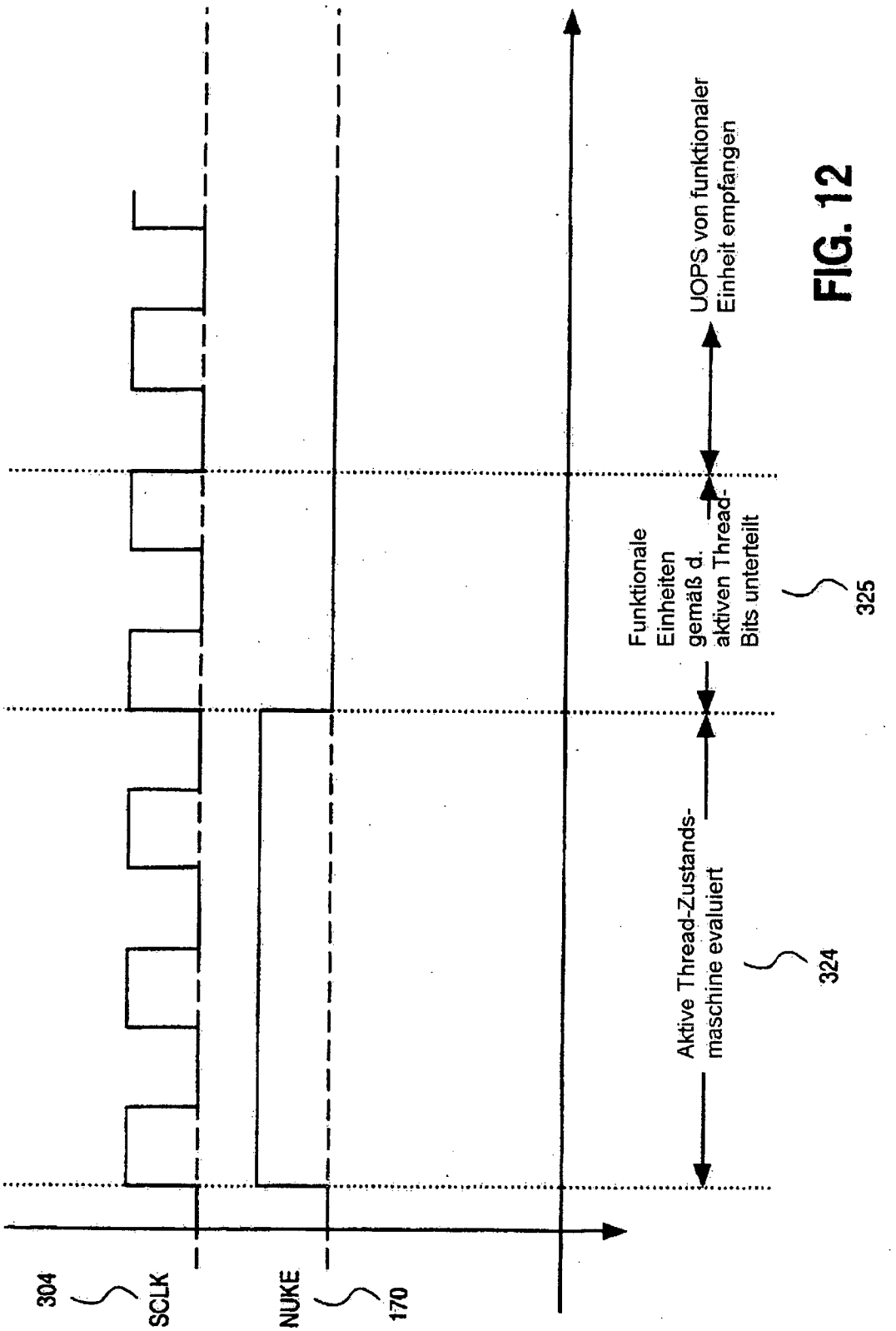

304
SCLK
NUKE
170
Aktive Thread-Zustands-maschine evaluiert
324
Funktionale Einheiten gemäß d. aktiven Thread-Bits unterteilt
325
UOPS von funktionaler Einheit empfangen

FIG. 12

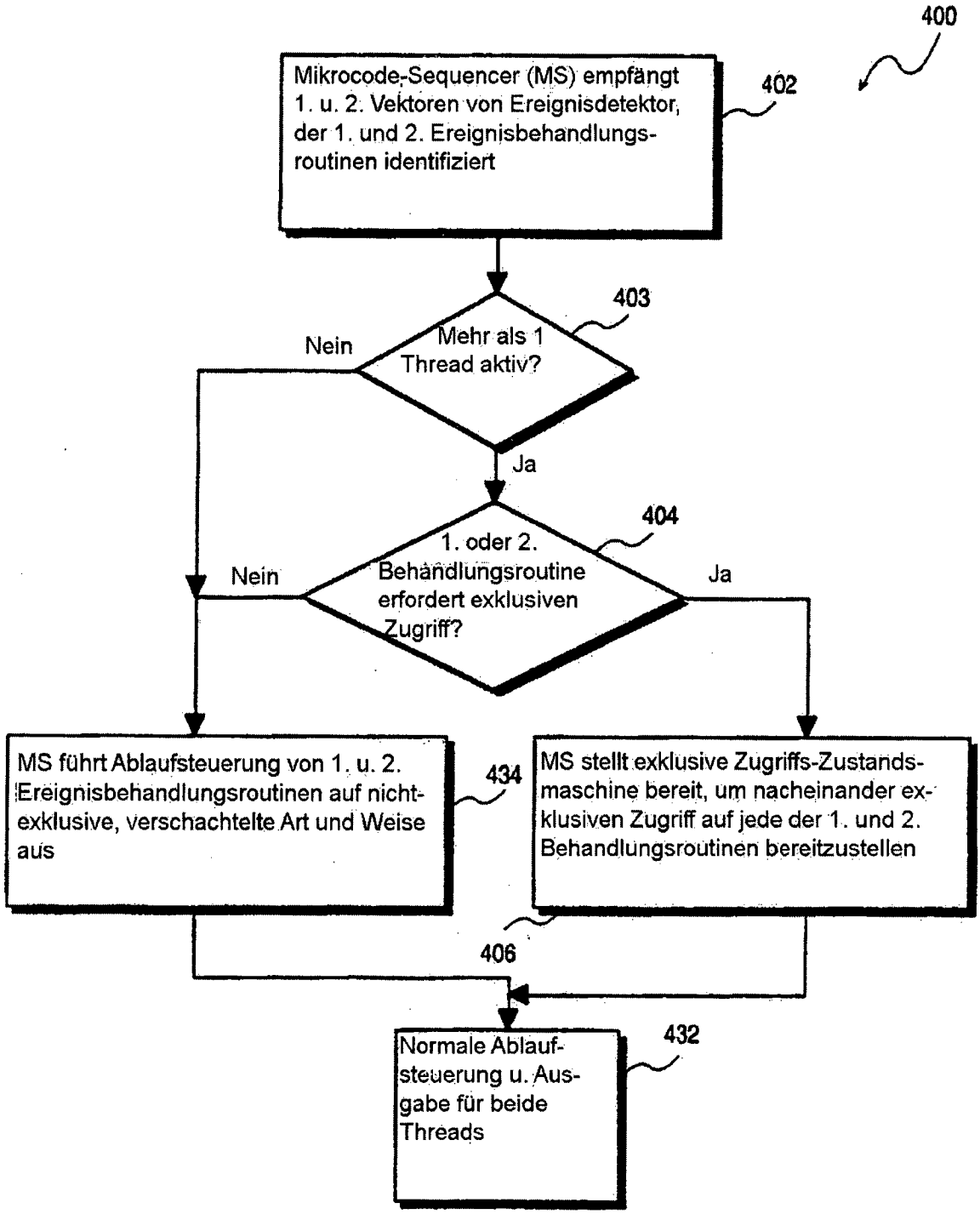

400
Mikrocode-Sequencer (MS) empfängt 1. u. 2. Vektoren von Ereignisdetektor, der 1. und 2. Ereignisbehandlungs-routinen identifiziert
402
403
Mehr als 1 Thread aktiv?
Nein
Ja
404
1. oder 2. Behandlungsroutine erfordert exklusiven Zugriff?
Nein
Ja
MS führt Ablaufsteuerung von 1. u. 2. Ereignisbehandlungsroutinen auf nicht-exklusive, verschachtelte Art und Weise aus
434
MS stellt exklusive Zugriffs-Zustands-maschine bereit, um nacheinander ex-klusiven Zugriff auf jede der 1. und 2. Behandlungsroutinen bereitzustellen
406
Normale Ablauf-steuerung u. Aus-gabe für beide Threads
432

FIG. 13

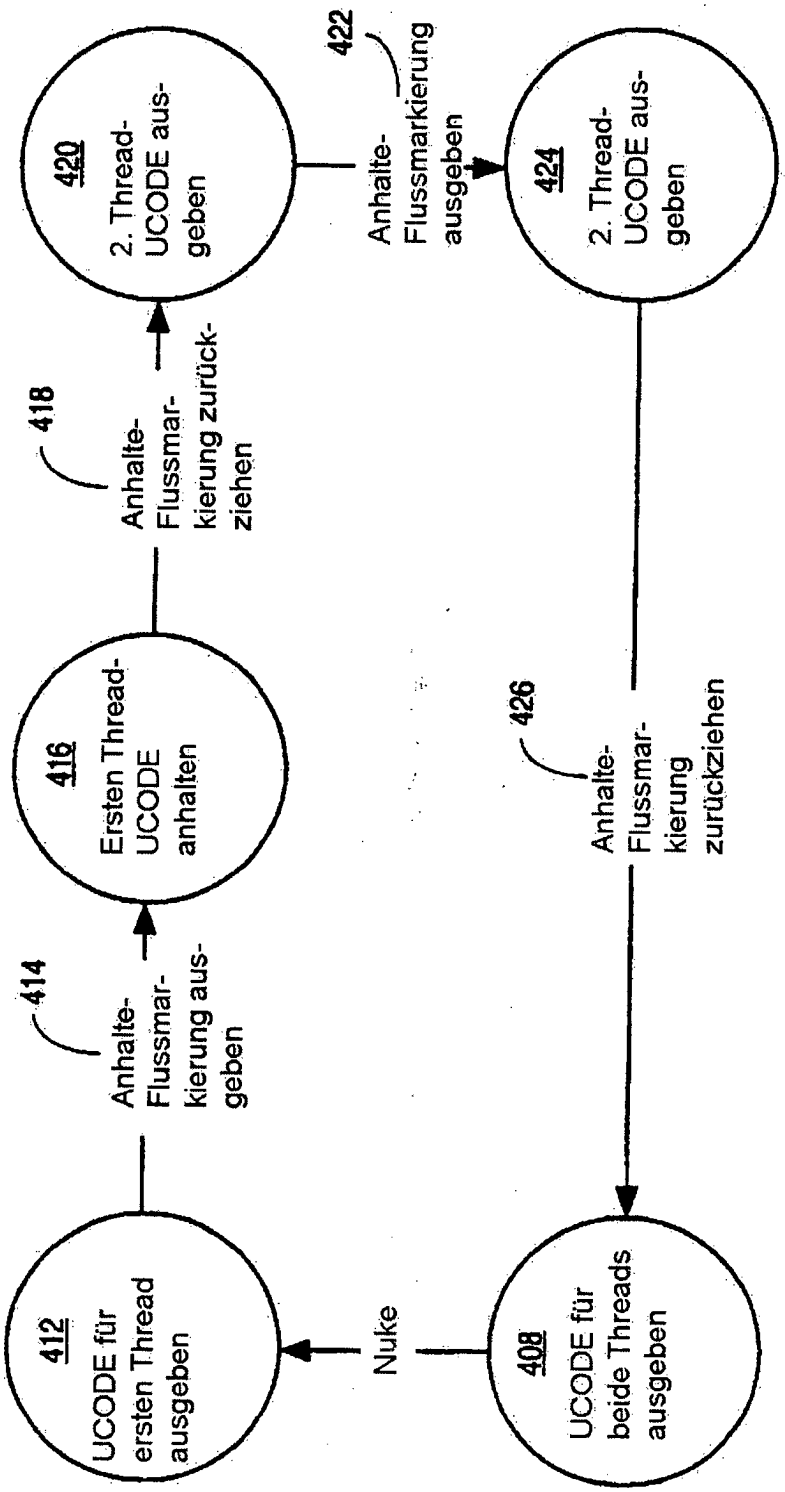

408
UCODE für beide Threads ausgeben
Nuke
412
UCODE für ersten Thread ausgeben
414
Anhalte-Flussmar-kierung aus-geben
416
Ersten Thread-UCODE anhalten
418
Anhalte-Flussmar-kierung zurück-ziehen
420
2. Thread-UCODE aus-geben
422
Anhalte-Flussmarkierung ausgeben
424
2. Thread-UCODE aus-geben
426
Anhalte-Flussmar-kierung zurückziehen

FIG. 14

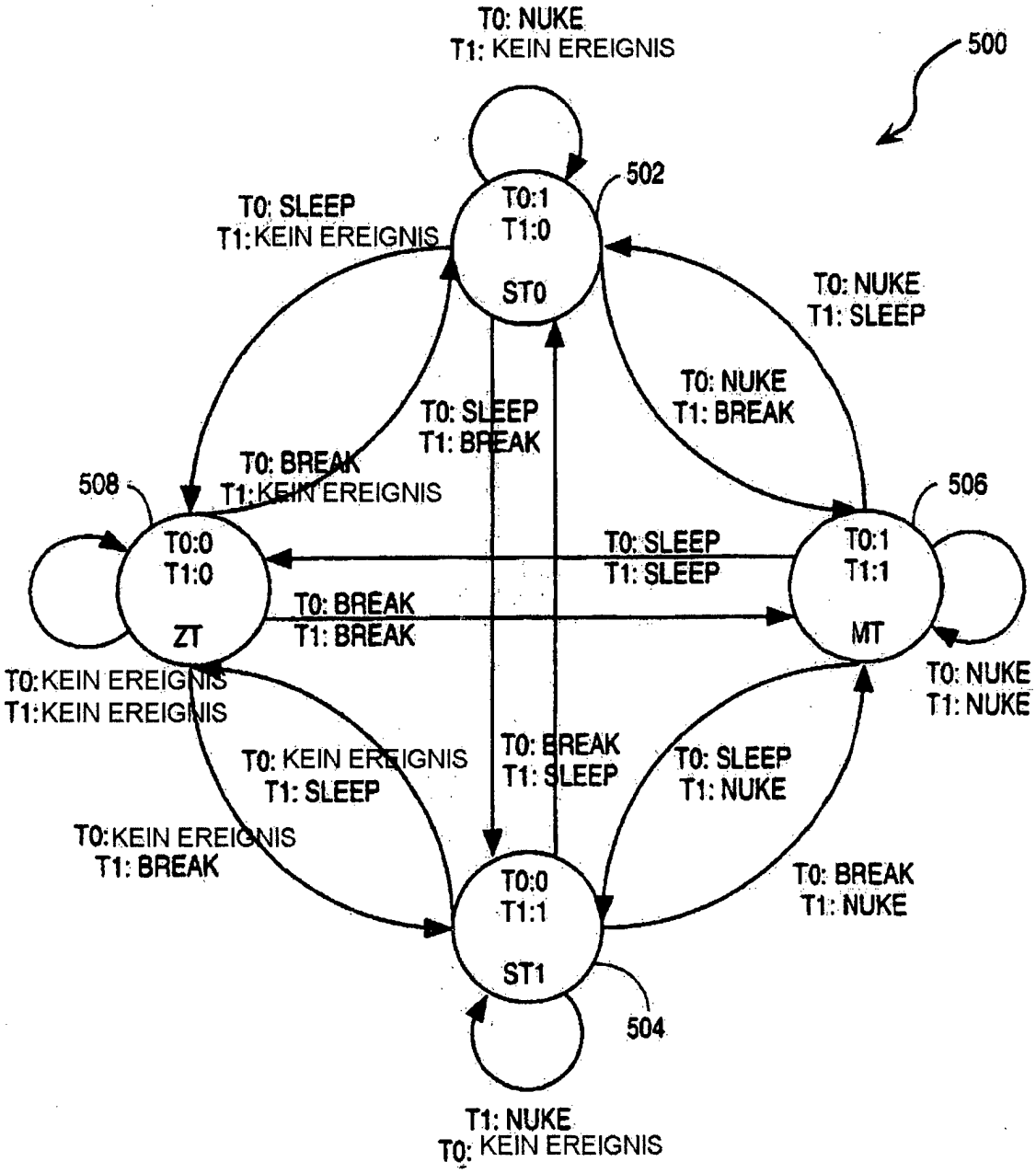

T0: NUKE
T1: KEIN EREIGNIS
500
502
T0:1
T1:0
ST0
T0: SLEEP
T1: KEIN EREIGNIS
T0: NUKE
T1: SLEEP
T0: NUKE
T1: BREAK
T0: SLEEP
T1: BREAK
T0: BREAK
T1: KEIN EREIGNIS
508
506
T0:0
T1:0
ZT
T0: SLEEP
T1: SLEEP
T0:1
T1:1
MT
T0: BREAK
T1: BREAK
T0: KEIN EREIGNIS
T1: KEIN EREIGNIS
T0: NUKE
T1: NUKE
T0: KEIN EREIGNIS
T1: SLEEP
T0: BREAK
T1: SLEEP
T0: SLEEP
T1: NUKE
T0: KEIN EREIGNIS
T1: BREAK
T0: BREAK
T1: NUKE
T0:0
T1:1
ST1
504
T1: NUKE
T0: KEIN EREIGNIS

FIG. 15

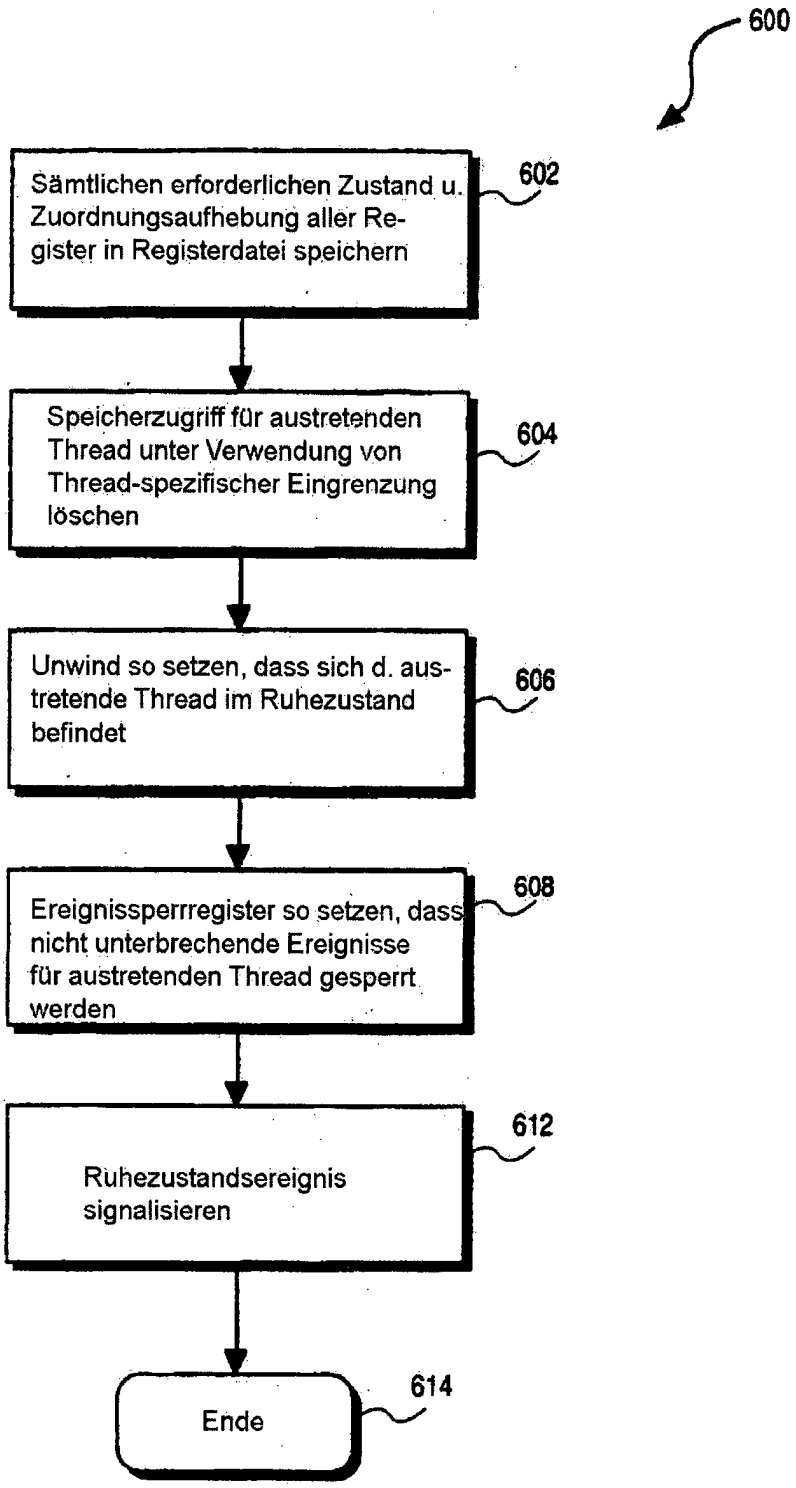

600
Sämtlichen erforderlichen Zustand u. Zuordnungsaufhebung aller Register in Registerdatei speichern
602
Speicherzugriff für austretenden Thread unter Verwendung von Thread-spezifischer Eingrenzung löschen
604
Unwind so setzen, dass sich d. austretende Thread im Ruhezustand befindet
606
Ereignissperrregister so setzen, dass nicht unterbrechende Ereignisse für austretenden Thread gesperrt werden
608
Ruhezustandsereignis signalisieren
612
Ende
614

FIG. 16A

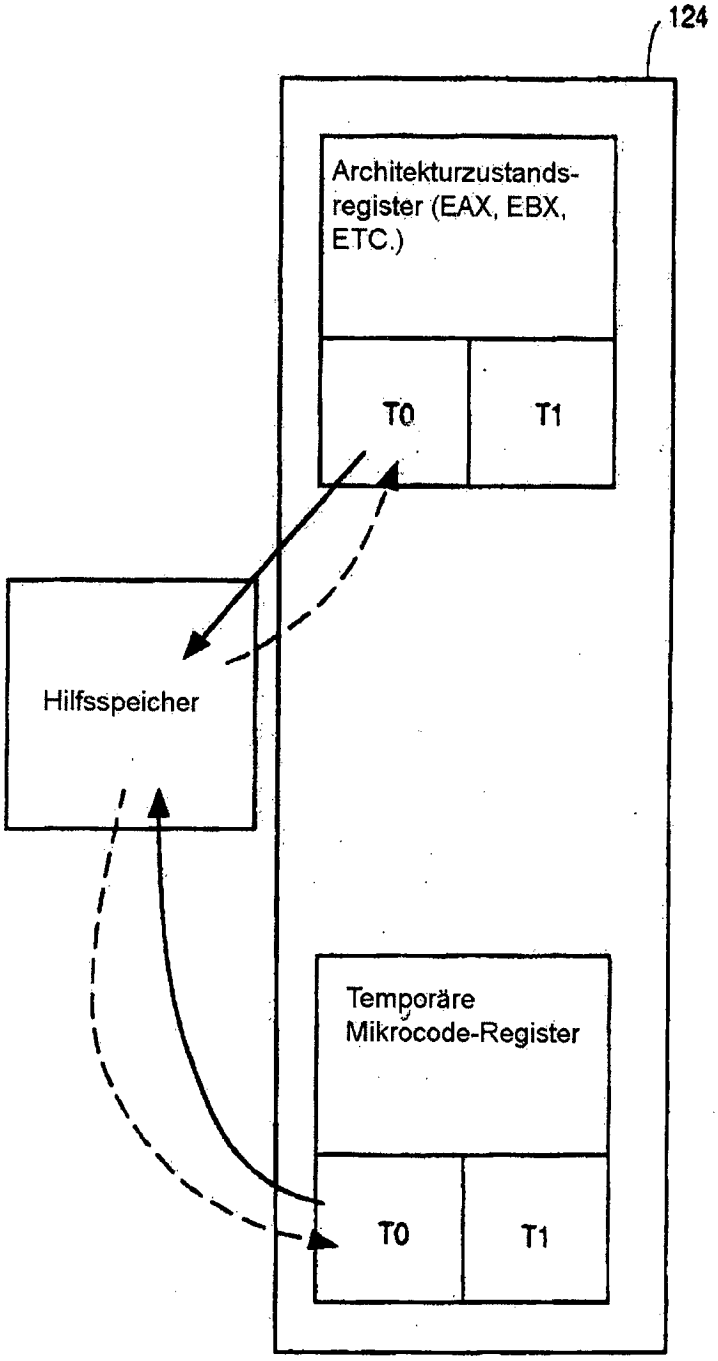

124
Architekturzustands-register (EAX, EBX, ETC.)
T0
T1
Hilfsspeicher
Temporäre Mikrocode-Register
T0
T1

FIG. 16B

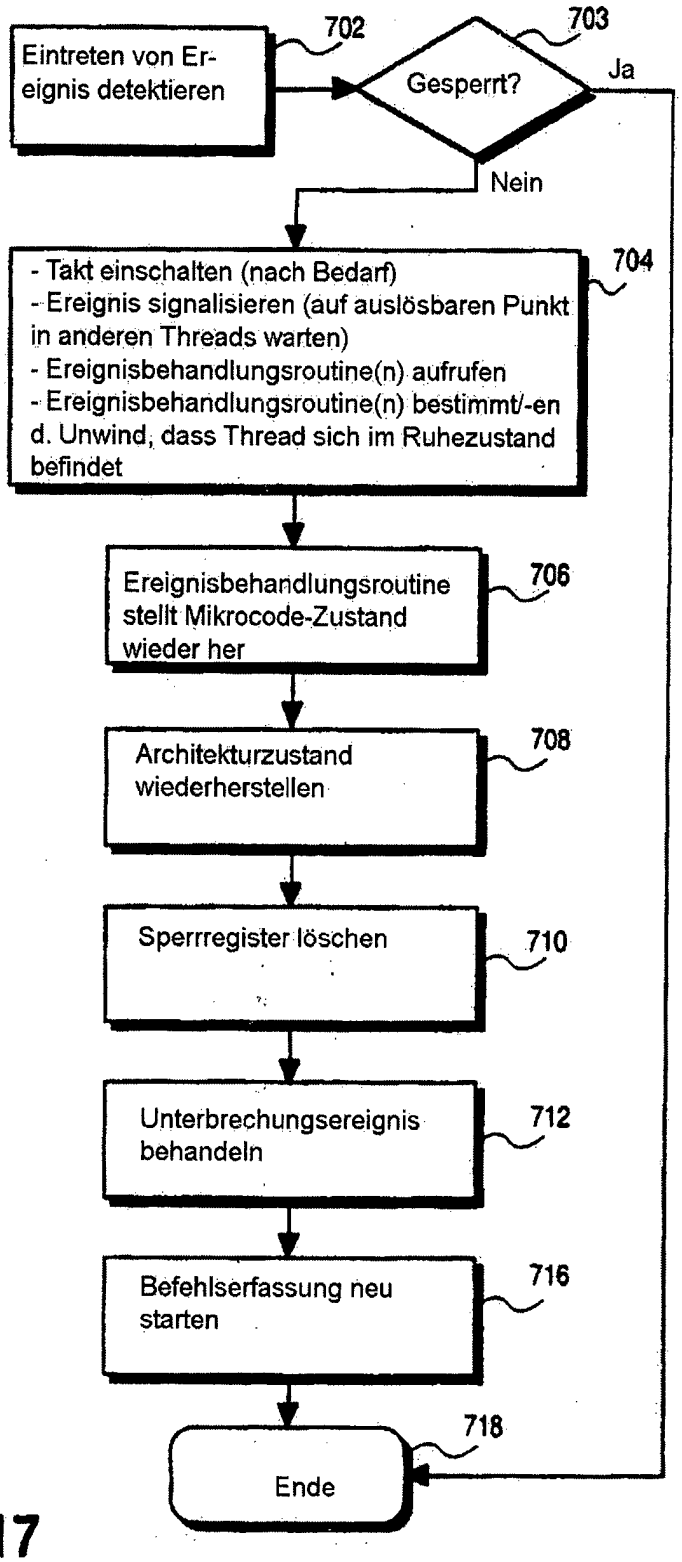

Eintreten von Ereignis detektieren
702
Gesperrt?
703
Ja
Nein
- Takt einschalten (nach Bedarf)
- Ereignis signalisieren (auf auslösbaren Punkt in anderen Threads warten)
- Ereignisbehandlungsroutine(n) aufrufen
- Ereignisbehandlungsroutine(n) bestimmt/-end. Unwind, dass Thread sich im Ruhezustand befindet
704
Ereignisbehandlungsroutine stellt Mikrocode-Zustand wieder her
706
Architekturzustand wiederherstellen
708
Sperrregister löschen
710
Unterbrechungsereignis behandeln
712
Befehlserfassung neu starten
716
Ende
718

FIG. 17

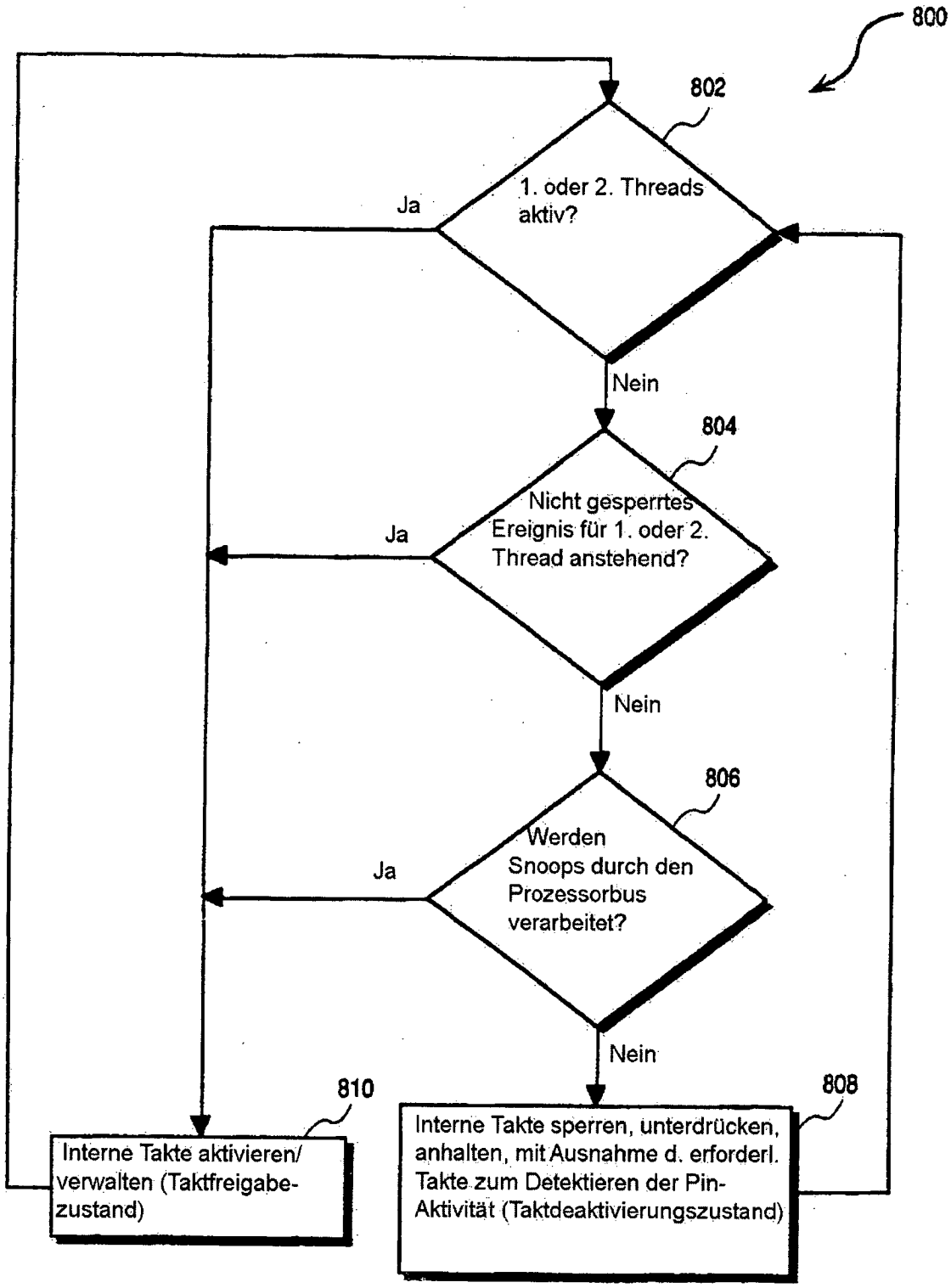

800
802
1. oder 2. Threads aktiv?
Ja
Nein
804
Nicht gesperrtes Ereignis für 1. oder 2. Thread anstehend?
Ja
Nein
806
Werden Snoops durch den Prozessorbus verarbeitet?
Ja
Nein
810
Interne Takte aktivieren/ verwalten (Taktfreigabe-zustand)
808
Interne Takte sperren, unterdrücken, anhalten, mit Ausnahme d. erforderl. Takte zum Detektieren der Pin-Aktivität (Taktdeaktivierungszustand)

FIG. 18

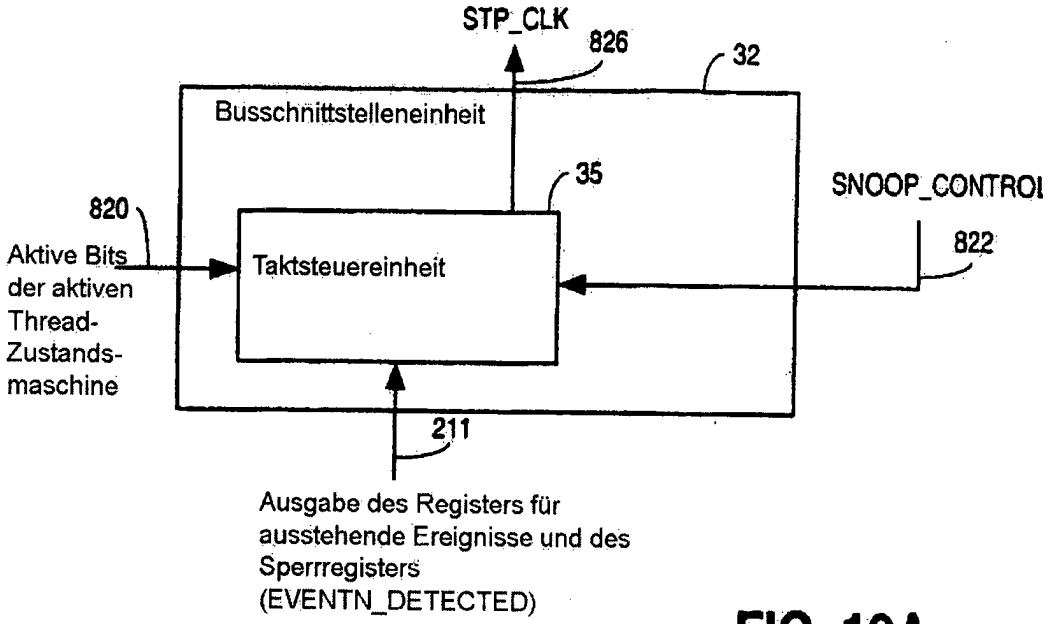


FIG. 19A

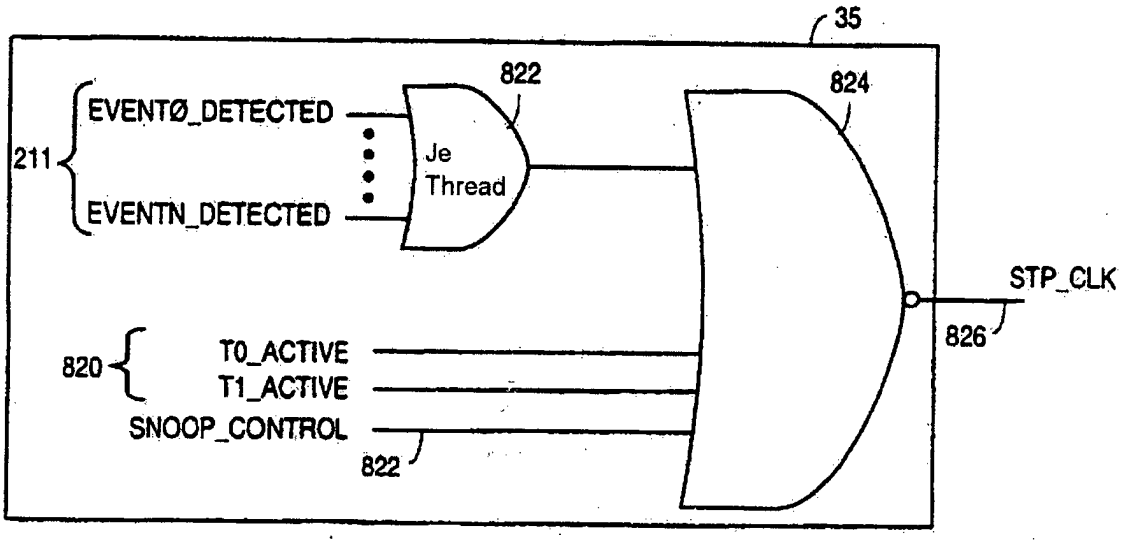


FIG. 19B