



(10) **DE 10 2018 126 004 A1** 2019.04.25

(12)

Offenlegungsschrift

(21) Aktenzeichen: **10 2018 126 004.0**

(22) Anmeldetag: **19.10.2018**

(43) Offenlegungstag: **25.04.2019**

(51) Int Cl.: **G06F 9/48 (2006.01)**

(30) Unionspriorität:

1717291.7 **20.10.2017** **GB**

(71) Anmelder:

Graphcore Limited, Bristol, GB

(74) Vertreter:

**Page, White & Farrer Germany LLP, 80538
München, DE**

(72) Erfinder:

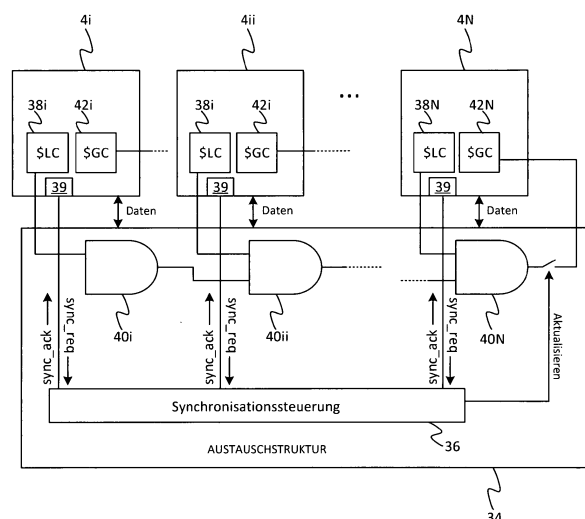
**Knowles, Simon Christian, Bath and North East
Somerset, GB; Alexander, Alan Graham, Wotton-
Under-Edge, GB**

Prüfungsantrag gemäß § 44 PatG ist gestellt.

Die folgenden Angaben sind den vom Anmelder eingereichten Unterlagen entnommen.

(54) Bezeichnung: **Synchronisation in einer Multi-Kachel-Verarbeitungsanordnung**

(57) Zusammenfassung: Verarbeitungssystem mit mehreren Kacheln und einer Zwischenverbindung zwischen den Kacheln. Die Zwischenverbindung wird verwendet zum Kommunizieren zwischen einer Gruppe aus einigen oder allen Kacheln entsprechend einem massensynchronen Parallelschema, wodurch jede Kachel in der Gruppe eine kachelinterne Rechenphase gefolgt von einer kachelübergreifenden Austauschphase durchführt, wobei die Austauschphase zurückgehalten wird, bis alle Kacheln in der Gruppe die Rechenphase beendet haben. Jede Kachel in der Gruppe weist bei der Beendigung der Rechenphase einen lokalen Ausstiegszustand auf. Der Befehlssatz umfasst einen Synchronisationsbefehl zur Ausführung durch jede Kachel bei der Beendigung ihrer Rechenphase zum Signalisieren einer Synchronisationsanforderung an eine Logik in der Zwischenverbindung. Im Ansprechen auf einen Empfang der Synchronisationsanforderung von allen Kacheln in der Gruppe gibt die Logik die nächste Austauschphase frei und macht auch einen zusammengefassten Zustand aller Kacheln in der Gruppe verfügbar.



Beschreibung

Technisches Gebiet

[0001] Die vorliegende Offenbarung bezieht sich auf eine Synchronisation der Arbeitslasten mehrerer verschiedener Kacheln (auch Tiles genannt) in einer Multi-Kachel-Verarbeitungsanordnung, wobei jede Kachel ihre eigene Verarbeitungseinheit und ihren eigenen Speicher aufweist. Insbesondere bezieht sich die Offenbarung auf Kommunikationsschemata zur massensynchronen Parallelverarbeitung (Bulk Synchronous Parallel (BSP)), wodurch jede aus einer Gruppe von Kacheln eine Rechenphase beenden muss, bevor irgendeine der Kacheln in der Gruppe zu einer Austauschphase fortschreiten kann.

Hintergrund

[0002] Ein Multi-Thread-Prozessor ist ein Prozessor, der mehrere Programm-Threads nebeneinander ausführen kann. Der Prozessor kann eine den mehreren verschiedenen Threads gemeinsame Hardware umfassen (z.B. einen gemeinsamen Befehlsspeicher, Datenspeicher und/oder Ausführungseinheit); um aber das Multi-Threading zu unterstützen, umfasst der Prozessor auch dedizierte Hardware, die für jeden Thread spezifisch ist.

[0003] Die dedizierte Hardware umfasst zumindest eine jeweilige Kontextregisterdatei für jeden der Anzahl der Threads, die gleichzeitig ausgeführt werden können. Im Zusammenhang mit Multi-Thread-Prozessoren ist unter einem „Kontext“ ein Programmzustand eines entsprechenden der parallel ausgeführten Threads zu verstehen (z.B. Programmzählerwert, Status und aktueller Operandenwert). Die Kontextregisterdatei bezieht sich auf die jeweilige Zusammenstellung von Registern zur Repräsentation dieses Programmzustands des jeweiligen Threads. Register in einer Registerdatei unterscheiden sich von einem Allzwecksspeicher dadurch, dass Registeradressen als Bits in Befehlsworten fixiert sind, wogegen Speicheradressen durch Ausführung von Befehlen berechnet werden können. Typischerweise umfassen Register eines gegebenen Kontexts einen jeweiligen Programmzähler für den jeweiligen Thread, und eine jeweilige Gruppe von Operandenregistern zum vorübergehenden Speichern der bearbeiteten Daten und Ausgabe durch den jeweiligen Thread während der durch diesen Thread durchgeführten Berechnungen. Jeder Kontext kann auch ein entsprechendes Statusregister zum Speichern eines Status des jeweiligen Threads aufweisen (z.B. ob er unterbrochen ist oder läuft). Somit weist jeder der gerade ablaufenden Threads seinen eigenen getrennten Programmzähler auf, und optional Befehlsregister und Statusregister.

[0004] Eine mögliche Form des Multi-Threading ist die Parallelverarbeitung. D.h., neben mehrfachen Kontexten sind auch mehrfache Ausführungspipelines vorgesehen: d.h., eine getrennte Ausführungspipeline für jeden Strom von parallel auszuführenden Befehlen. Dies erfordert jedoch ein hohes Maß an Duplizierung in Bezug auf die Hardware.

[0005] Daher wird in einer anderen Form von Multi-Thread-Prozessor eine Gleichzeitigkeit anstelle der Parallelität eingesetzt, wodurch die Threads eine gemeinsame Ausführungspipeline (oder zumindest einen gemeinsamen Teil einer Pipeline) teilen und verschiedene Threads durch diese gleiche, gemeinsam genutzte Ausführungspipeline verschachtelt sind. Die Leistungsfähigkeit eines Multi-Thread-Prozessors kann im Vergleich zu keiner Gleichzeitigkeit oder Parallelität dennoch verbessert werden, dank verbesserter Möglichkeiten zur Verbergung der Pipelinelatenz. Auch erfordert dieser Ansatz nicht so viel dedizierte Zusatz-Hardware für jeden Thread, wie bei einem Vollparallelprozessor mit mehrfachen Ausführungspipelines, und erfordert daher nicht so viel zusätzliches Silizium.

[0006] Eine Form der Parallelität kann erreicht werden mittels eines Prozessors mit einer Anordnung von mehreren Kacheln auf demselben Chip (d.h. demselben Chip-Plättchen (Die)), wobei jede Kachel ihre jeweilige eigene getrennte Verarbeitungseinheit mit Speicher (inklusive Programmspeicher und Datenspeicher) aufweist. Somit können getrennte Teile des Programmcodes parallel auf verschiedenen Kacheln ausgeführt werden. Die Kacheln werden über eine chipinterne Zwischenverbindung miteinander verbunden, wodurch der auf den verschiedenen Kacheln ausgeführte Code zwischen den Kacheln kommunizieren kann. In einigen Fällen kann die Verarbeitungseinheit auf jeder Kachel selbst mehrere gleichzeitige Threads auf der Kachel ausführen, wobei jede Kachel ihre eigene jeweilige Gruppe von Kontexten mit entsprechender Pipeline aufweisen kann, wie vorstehend beschrieben, um eine Verschachtelung mehrerer Threads auf derselben Kachel durch dieselbe Pipeline zu unterstützen.

[0007] Im Allgemeinen können Abhängigkeiten zwischen den Teilen eines auf verschiedenen Kacheln ablaufenden Programms existieren. Es ist daher eine Technik erforderlich, die verhindert, dass ein Codeabschnitt auf einer Kachel vorseilt gegenüber Daten, von denen er abhängig ist und die verfügbar gemacht werden durch einen anderen Codeabschnitt auf einer anderen Kachel. Zu diesem Zweck gibt es eine Vielzahl möglicher Maßnahmen, wobei aber die hier interessierende Maßnahme als „massensynchrone Parallelverarbeitung“ (BSP) bekannt ist. Gemäß BSP führt jede Kachel in einem abwechselnden Zyklus eine Rechenphase und eine Austauschphase aus. Während der Rechenphase führt jede Kachel eine oder mehrere Berechnungsaufgaben lokal auf der Kachel aus, kommuniziert aber keine Ergebnisse dieser Berechnungen mit anderen Kacheln. In der Austauschphase kann jede Kachel ein oder mehrere Ergebnisse der Berechnungen aus der vorherigen Rechenphase zu und/oder von einer oder mehreren anderen Kacheln der Gruppe austauschen, schreitet aber noch nicht zur nächsten Rechenphase fort. Ferner wird gemäß dem BSP-Prinzip eine Barriersynchronisation an der von der Rechenphase zur Austauschphase oder von der Austauschphase zur Rechenphase, oder beidem übergehenden Verbindungsstelle platziert. D.h., entweder: (a) alle Kacheln müssen ihre jeweiligen Rechenphasen abgeschlossen haben, bevor eine aus der Gruppe zur nächsten Austauschphase fortschreiten darf, oder (b) alle Kacheln in der Gruppe müssen ihre jeweiligen Austauschphasen abgeschlossen haben, bevor eine beliebige Kachel der Gruppe zur nächsten Rechenphase fortschreiten darf, oder (c) beides. In einigen Szenarien kann es einer Kachel, die eine Berechnung durchführt, erlaubt sein, mit anderen Systemressourcen wie beispielsweise einer Netzwerkkarte oder einer Speicherplatte zu kommunizieren, solange keine Kommunikation mit anderen Kacheln aus der Gruppe involviert ist.

[0008] Ein Beispiel für die Verwendung der Multi-Thread- und/oder Multi-Kachel-Verarbeitung ist die Maschinenintelligenz. Wie der Fachmann auf dem Gebiet der Maschinenintelligenz weiß, basiert ein Maschinenintelligenzalgorithmus auf der Durchführung iterativer Aktualisierungen eines „Wissensmodells“, was durch einen Graphen aus mehrfach verbundenen Knoten dargestellt werden kann. Jeder Knoten repräsentiert eine Funktion seiner Eingänge. Einige Knoten empfangen die Eingaben des Graphen und andere empfangen Eingaben von einem oder mehreren anderen Knoten, während die Ausgabe einiger Knoten die Eingaben anderer Knoten bilden, und die Ausgabe einiger Knoten die Ausgabe des Graphen bereitstellen (und in einigen Fällen kann ein gegebener Knoten sogar alles davon aufweisen: Eingaben in den Graphen, Ausgaben von dem Graphen und Verbindungen zu anderen Knoten). Des Weiteren wird die Funktion eines jeden Knoten durch einen oder mehrere jeweilige Parameter, z.B. Gewichtungen, parametrisiert. Während einer Lernphase besteht das Ziel darin, basierend auf einer Gruppe experimenteller Eingangsdaten Werte für die verschiedenen Parameter aufzufinden, sodass der Graph als Ganzes eine gewünschte Ausgabe für einen Bereich möglicher Eingaben erzeugt. Verschiedene Algorithmen dafür sind im Stand der Technik bekannt, wie beispielsweise ein Rückausbreitungsalgorithmus basierend auf einem stochastischen Gradientenabstieg. Über mehrere Iterationen werden die Parameter basierend auf den Eingangsdaten allmählich zur Verringerung ihrer Fehler eingestellt, und der Graph konvergiert somit in Richtung einer Lösung. In einer nachfolgenden Phase kann das gelernte Modell verwendet werden zur Durchführung von Vorhersagen für Ausgaben bei einem vorgegebenen spezifischen Satz von Eingaben oder zum Ableiten von Eingaben (Ursachen) bei einem vorgegebenen spezifischen Satz von Ausgaben.

[0009] Die Implementierung eines jeden Knoten involviert die Verarbeitung von Daten und die Zwischenverbindungen des Graphen entspricht den zwischen den Knoten auszutauschenden Daten. Typischerweise kann zumindest ein Teil der Verarbeitung eines jeden Knoten unabhängig von einigen oder allen anderen Knoten in dem Graphen ausgeführt werden, und daher bieten große Graphen überragende Möglichkeiten für Gleichzeitigkeit und/oder Parallelität.

Zusammenfassung

[0010] Im Folgenden werden Komponenten eines Prozessors mit einer Architektur beschrieben, die entwickelt wurde zum Adressieren von sich bei den in Maschinenintelligenzanwendungen involvierten Berechnungen ergebenden Problemen. Der hier beschriebene Prozessor kann als ein Arbeitsbeschleuniger verwendet werden, das heißt, er empfängt eine Arbeitslast von einer auf einem Hostcomputer ablaufenden Anwendung, wobei die Arbeitslast generell die Form sehr großer zu verarbeitender Datensätze aufweist (wie beispielsweise die von einem Maschinenintelligenzalgorithmus verwendeten umfangreichen Experimentierdatensätze zum Erlernen eines Wissensmodells, oder die Daten, aus denen eine Vorhersage oder Inferenz unter Verwendung eines zuvor erlernten Wissensmodells erfolgt). Ein Ziel der hier vorgestellten Architektur liegt im hocheffizienten Verarbeiten dieser sehr großen Datenmengen. Die Prozessorarchitektur wurde zur Verarbeitung von bei der Maschinenintelligenz involvierten Arbeitslasten entwickelt. Dennoch ist ersichtlich, dass die offenbarte Architektur auch für andere, ähnliche Charakteristiken teilende Arbeitslasten geeignet ist.

[0011] Bei der Ausführung verschiedener Abschnitte eines Programms über mehrere Kacheln kann es erforderlich sein, eine Barriersynchronisation durchzuführen, um mehrere Kacheln auf einen gemeinsamen Ausführungspunkt zu bringen. Auch kann es wünschenswert sein, einen Zustand des Programms als Ganzes nach der Beendigung einer Rechenphase durch alle Kacheln zu bestimmen, um beispielsweise festzustellen, ob oder nicht ein Ausnahmezustand dem Host berichtet werden sollte, oder um eine Abzweigungsentscheidung zu treffen, um festzustellen, ob zu einem nächsten Abschnitt des Programms abgezweigt werden soll oder der aktuelle Abschnitt wiederholt fortgeführt werden soll. Falls beispielsweise jede aus einer Gruppe von Kacheln die Berechnungen eines entsprechenden Untergraphen eines Maschinenintelligenzgraphen durchführt, kann es wünschenswert sein, festzustellen, ob alle Knoten des Untergraphen eine bestimmte Bedingung erfüllt haben, die angibt, dass der Graph in Richtung einer Lösung konvergiert. Um eine solche Feststellung unter Verwendung existierender Techniken zu treffen, sind eine Vielzahl von Schritten erforderlich, die unter Verwendung von Allzweckbefehlen programmiert sind.

[0012] Es ist hier ersichtlich, dass es wünschenswert wäre, den Befehlssatz eines Prozessors auf umfangreiche Multi-Thread-Anwendungen, wie beispielsweise Maschinenlernen, abzustimmen. Gemäß der vorliegenden Offenbarung wird dies erreicht durch Bereitstellen eines dedizierten Maschinencodebefehls zum Validieren eines Ergebnisses einer Gruppe von Kacheln erst dann, wenn alle Kacheln in der Gruppe die aktuelle BSP-Rechenphase beendet haben, sodass die Kacheln synchronisiert werden können und dabei ein Gesamtergebnis der mehreren Threads mit verringerter Latenz und geringerer Codedichte bestimmt wird.

[0013] Gemäß einem hier offenbarten Aspekt wird ein Verarbeitungssystem bereitgestellt mit einer Anordnung von Kacheln und einer Zwischenverbindung zur Kommunikation zwischen den Kacheln, wobei:

jede Kachel eine Ausführungseinheit aufweist zum Ausführen von Maschinencodebefehlen, wobei jeder eine Instanz einer vordefinierten Gruppe von Befehlstypen in einem Befehlssatz des Prozessors ist, wobei jeder Befehlstyp in dem Befehlssatz durch einen entsprechenden Befehlscode und keine oder mehr Operandenfelder zur Aufnahme von keinem oder mehr Operanden definiert ist;

die Zwischenverbindung betreibbar ist zum Durchführen von Kommunikationen zwischen einer Gruppe aus einigen oder allen Kacheln gemäß einem massensynchronen Parallelschema, wodurch jede der Kacheln in der Gruppe eine kachelinterne Rechenphase gefolgt von einer kachelübergreifenden Austauschphase durchführt, wobei die Austauschphase zurückgehalten wird, bis alle Kacheln in der Gruppe die Rechenphase abgeschlossen haben, wobei jede Kachel in der Gruppe bei der Beendigung der Rechenphase einen lokalen Ausstiegzzustand aufweist;

der Befehlssatz einen Synchronisationsbefehl zur Ausführung durch jede Kachel in der Gruppe beim Beenden ihrer Rechenphase umfasst, wobei eine Ausführung des Synchronisationsbefehls die Ausführungseinheit veranlasst, eine Synchronisationsanforderung an eine Hardwarelogik in der Zwischenverbindung zu senden; und

die Logik in der Zwischenverbindung ausgestaltet ist zum Zusammenfassen der lokalen Ausstiegzzustände in einen globalen Ausstiegzzustand, und zum Speichern des globalen Ausstiegzzustands in einem Globalausstiegzzustandsregister auf jeder der Kacheln in der Gruppe im Ansprechen auf die Beendigung der Rechenphase durch alle Kacheln in der Gruppe, was angezeigt wird durch einen Empfang der Synchronisationsanforderung von allen Kacheln in der Gruppe, sodass der globale Ausstiegzzustand durch einen auf jeder der Kacheln in der Gruppe ablaufenden Codeabschnitt zugänglich ist.

[0014] Gemäß einem weiteren hier offenbarten Aspekt wird ein Verarbeitungssystem mit einer Anordnung von Kacheln und einer Zwischenverbindung zur Kommunikation zwischen den Kacheln bereitgestellt, wobei:

jede Kachel eine entsprechende Ausführungseinheit zum Ausführen von Maschinencodebefehlen umfasst, wobei jeder eine Instanz einer vordefinierten Gruppe von Befehlstypen in einem Befehlssatz des Prozessors ist, wobei jeder Befehlstyp in dem Befehlssatz durch einen entsprechenden Befehlscode und keine oder mehr Operandenfelder zum Aufnehmen von keinem oder mehr Operanden definiert ist;

die Zwischenverbindung eine Synchronisationslogik als dedizierte Hardwarelogik zur Koordination zwischen einer Gruppe von einigen oder allen der Kacheln umfasst;

der Befehlssatz einen Synchronisationsbefehl umfasst, wobei die entsprechende Ausführungseinheit auf jeder entsprechenden Kachel so ausgestaltet ist, dass dann, wenn eine Instanz des Synchronisationsbefehls durch die entsprechende Ausführungseinheit ausgeführt wird, eine Instanz einer Synchronisationsanforderung im Ansprechen auf den Befehlscode des Synchronisationsbefehls von der entsprechenden Kachel zu der Synchronisationslogik in der Zwischenverbindung gesendet wird, und eine Befehlsausgabe

auf der entsprechenden Kachel unterdrückt wird, bis eine Synchronisationsbestätigung von der Synchronisationslogik zurückempfungen wird;

jede Kachel ein Register mit einem Lokalausstiegszustandsregister aufweist zum Speichern eines lokalen Ausstiegszustands der Kachel beim Beenden einer jeweiligen Rechenphase;

die Synchronisationslogik ausgestaltet ist zum Zusammenfassen der lokalen Ausstiegszustände der Kacheln in der Gruppe zu einem globalen Ausstiegszustand; und

die Synchronisationslogik ferner so ausgestaltet ist, dass sie im Ansprechen auf einen Empfang einer Instanz der Synchronisationsanforderung von allen Kacheln der Gruppe die Synchronisationsbestätigung zu jeder Kachel in der Gruppe zurücksendet und dadurch eine Wiederaufnahme der Befehlsausgabe ermöglicht, und den globalen Ausstiegszustand in einem Globalausstiegszustandsregister auf jeder Kachel in der Gruppe speichert, sodass der globale Ausstiegszustand durch einen auf jeder der Kacheln in der Gruppe ablaufenden Codeabschnitt zugänglich ist.

[0015] In Ausführungsbeispielen kann die Ausführungseinheit auf jeder Kachel ausgestaltet sein zum Pausieren der Befehlsausgabe im Ansprechen auf ein Ausführen des Synchronisationsbefehls; und die Logik in der Zwischenverbindung kann so ausgestaltet sein, dass sie ein Synchronisationsbestätigungssignal im Ansprechen auf einen Empfang der Synchronisationsanforderung von allen Kacheln in der Gruppe an jede der Kacheln in der Gruppe zurücksendet, um die Befehlsausgabe wiederaufzunehmen.

[0016] In Ausführungsbeispielen kann jeder der lokalen Ausstiegszustände und der globalen Ausstiegszustände ein einzelnes Bit sein.

[0017] In Ausführungsbeispielen kann die Zusammenfassung aus einem Booleschen UND der lokalen Ausstiegszustände oder einem Booleschen ODER der lokalen Ausstiegszustände bestehen.

[0018] In alternativen Ausführungsbeispielen kann der zusammengefasste Ausstiegszustand zumindest zwei Bits umfassen, die einen trinären Wert repräsentieren, der angibt, ob die lokalen Ausstiegszustände alle wahr, alle falsch oder gemischt sind.

[0019] In Ausführungsbeispielen kann jede der Gruppe von Kacheln ein Lokalausstiegszustandsregister umfassen, das zum Repräsentieren des lokalen Ausstiegszustands der Kachel angeordnet ist.

[0020] In Ausführungsbeispielen kann jede Kachel in der Gruppe umfassen:

mehrere Kontextregistergruppen, wobei jede Kontextregistergruppe angeordnet ist zum Speichern eines Programmzustands eines entsprechenden aus einer Vielzahl von Threads; und

einen Scheduler ausgestaltet zum Steuern des Ablaufs der Ausführung eines jeweiligen aus einer Vielzahl von Worker-Threads in jedem aus einer Vielzahl von Zeitschlitten in einer wiederholten Sequenz von verschachtelten Zeitschlitten, wobei der Programmzustand eines jeden der Worker-Threads in einer entsprechenden der Kontextregistergruppen gespeichert ist;

wobei die Austauschphase gemäß dem massensynchronen Parallelschema zurückgehalten ist, bis alle Worker-Threads auf allen Kacheln in der Gruppe die Rechenphase beendet haben;

wobei der lokale Ausstiegszustand auf jeder Kachel eine Zusammenfassung eines individuellen von jedem der Worker-Threads auf der Kachel ausgegebenen Ausstiegszustands sein kann; und

wobei der Codeabschnitt zumindest einen der mehreren Threads auf der Kachel umfassen kann.

[0021] In Ausführungsbeispielen kann jede Kachel in der Gruppe eine Hardwarelogik aufweisen, die ausgestaltet ist zum Durchführen der Zusammenfassung der individuellen Ausstiegszustände zu dem lokalen Ausstiegszustand.

[0022] In Ausführungsbeispielen kann der Befehlssatz einen Ausstiegsbefehl umfassen zum Aufnehmen in jeden der Worker-Threads, und die Ausführungseinheit kann ausgestaltet sein zum Ausgeben des individuellen Ausstiegszustands des entsprechenden Worker-Threads und Beenden des entsprechenden Worker-Threads im Ansprechen auf den Befehlscode des Ausstiegsbefehls.

[0023] In Ausführungsbeispielen kann jeder der individuellen Ausstiegszustände und der lokalen Ausstiegszustände ein einzelnes Bit sein, und die Zusammenfassung der individuellen Ausstiegszustände kann aus ei-

nem Booleschen UND der individuellen Ausstiegzzustände oder einem Booleschen ODER der individuellen Ausstiegzzustände bestehen.

[0024] In Ausführungsbeispielen kann der lokale Ausstiegzzustand zumindest zwei einen trinären Wert repräsentierende Bits umfassen, der angibt, ob die individuellen Ausstiegzzustände alle wahr, alle falsch oder gemischt sind.

[0025] In Ausführungsbeispielen kann die Austauschphase so ausgestaltet sein, dass sie von einem Supervisor-Thread getrennt von den Worker-Threads ausgeführt wird, und der zumindest eine Thread kann den Supervisor-Thread umfassen.

[0026] In Ausführungsbeispielen kann das Pausieren der Befehlsausgabe zumindest ein Pausieren der Befehlsausgabe von dem Supervisor-Thread umfassen, bis zur Synchronisationsbestätigung.

[0027] In Ausführungsbeispielen können die Kontextregistergruppen auf jeder Kachel umfassen: mehrere Arbeiterkontextregistergruppen angeordnet zum Repräsentieren des Programmzustands entsprechender der Vielzahl von Worker-Threads, und eine zusätzliche Supervisor-Kontextregistergruppe mit einer zusätzlichen Gruppe von Registern angeordnet zum Repräsentieren eines Programmzustands des Supervisor-Threads.

[0028] In Ausführungsbeispielen:

kann der Supervisor-Thread so ausgestaltet sein, dass er durch Ablaufen in jedem der Zeitschlitzte beginnt;

kann der Befehlssatz ferner einen Freigabebefehl umfassen und die Ausführungseinheit kann so ausgestaltet sein, dass sie im Ansprechen auf den Befehlscode des Freigabebefehls den Zeitschlitz, in dem der Freigabebefehl ausgeführt wird, an den entsprechenden Worker-Thread freigibt; und

kann der Ausstiegssbefehl eine Zurückgabe des entsprechenden Zeitschlitzes, in dem der Ausstiegssbefehl ausgeführt wird, an den Supervisor-Thread veranlassen, sodass der Supervisor-Thread den Ablauf in dem entsprechenden Schlitz wiederaufnimmt.

[0029] In Ausführungsbeispielen kann der Codeabschnitt ausgestaltet sein zum Verwenden des globalen Ausstiegzzustands, sobald dieser gültig ist, zum Durchführen einer von dem globalen Ausstiegzzustand abhängigen Abzweigungsentscheidung.

[0030] In Ausführungsbeispielen kann das Verarbeitungssystem programmiert sein zum Durchführen eines Maschinenintelligenzalgorithmus, bei dem jeder Knoten in einem Graphen eine oder mehrere jeweilige Eingangskanten und eine oder mehrere jeweilige Ausgangskanten aufweist, wobei die Eingangskanten von zumindest einigen der Knoten den Ausgangskanten von zumindest einigen anderen Knoten entsprechen, wobei jeder Knoten eine jeweilige seine Ausgangskanten mit seinen Eingangskanten in Beziehung bringende Funktion aufweist, wobei jede jeweilige Funktion durch einen oder mehrere jeweilige Parameter parametrisiert ist, und wobei jeder der jeweiligen Parameter einen verknüpften Fehler aufweist, sodass der Graph mit abnehmenden Fehlern in einigen oder allen Parametern in Richtung einer Lösung konvergiert, wobei jede der Kacheln einen entsprechenden Untergraphen mit einer Untergruppe der Knoten in dem Graphen repräsentiert, und jeder der lokalen Ausstiegzzustände verwendet werden kann zum Anzeigen, ob die Fehler in dem einen oder mehreren Parametern des Knoten in dem entsprechenden Untergraphen eine vorbestimmte Bedingung erfüllt haben.

[0031] In Ausführungsbeispielen kann die Gruppe zumindest teilweise durch einen Operanden des Synchronisationsbefehls ausgewählt werden.

[0032] In Ausführungsbeispielen kann der Operand des Synchronisationsbefehls auswählen, ob ausschließlich Kacheln auf demselben Chip oder Kacheln auf verschiedenen Chips in der Gruppe eingeschlossen sind.

[0033] In Ausführungsbeispielen kann der Operand des Synchronisationsbefehls die Gruppe aus verschiedenen hierarchischen Gruppierungsebenen auswählen.

[0034] In Ausführungsbeispielen kann der Befehlssatz ferner einen Verzichtsbefehl enthalten, der die Kachel, auf der der Verzichtsbefehl ausgeführt wird, zur Nichtteilnahme an der Gruppe veranlasst.

[0035] Gemäß einem weiteren hier offenbarten Aspekt wird ein Verfahren zum Betreiben eines Verarbeitungssystems mit einer Anordnung von Kacheln und einer Zwischenverbindung zur Kommunikation zwischen den Kacheln bereitgestellt, wobei jede Kachel eine Ausführungseinheit zum Ausführen von Maschinencodebefehl-

len aufweist, wobei jeder eine Instanz einer vordefinierten Gruppe von Befehlstypen in einem Befehlssatz des Prozessors ist, wobei jeder Befehlstyp in dem Befehlssatz durch einen entsprechenden Befehlscode und keine oder mehr Operandenfelder zur Aufnahme von keinem oder mehr Operanden definiert ist; wobei das Verfahren umfasst:

[0036] Durchführen von Kommunikationen zwischen einer Gruppe von einigen oder allen der Kacheln, über die Zwischenverbindung, gemäß einem massensynchronen Parallelschema, wodurch jede der Kacheln in der Gruppe eine kachelinterne Rechenphase gefolgt von einer kachelübergreifenden Austauschphase durchführt, wobei die Austauschphase zurückgehalten wird, bis alle Kacheln in der Gruppe die Rechenphase beendet haben, wobei jede Kachel in der Gruppe beim Beenden der Rechenphase einen lokalen Ausstiegsszustand aufweist;

wobei der Befehlssatz einen Synchronisationsbefehl zum Ausführen durch jede Kachel in der Gruppe beim Beenden ihrer Rechenphase umfasst, wobei ein Ausführen des Synchronisationsbefehls die Verarbeitungseinheit veranlasst, eine Synchronisationsanforderung an eine Hardwarelogik in der Zwischenverbindung zu senden; und

wobei das Verfahren ein Triggern der Logik in der Zwischenverbindung zum Zusammenfassen der lokalen Ausstiegsszustände zu einem globalen Ausstiegsszustand und zum Speichern des globalen Ausstiegsszustands in einem Globalausstiegsszustandsregister auf jeder der Kacheln in der Gruppe umfasst, im Ansprechen auf die Beendigung der Rechenphase durch alle Kacheln in der Gruppe, was durch einen Empfang der Synchronisationsanforderung von allen Kacheln in der Gruppe angezeigt wird, sodass der globale Ausstiegsszustand von einem auf jeder der Kacheln in der Gruppe ablaufenden Codeabschnitt zugänglich ist.

[0037] Gemäß einem weiteren hier offenbarten Aspekt wird ein Computerprogrammprodukt bereitgestellt mit einem auf einem Computer lesbaren Speicher verkörpertem Code und das ausgestaltet ist zum Ausführen auf dem Verarbeitungssystem eines jeden hier offenbarten Ausführungsbeispiels, wobei der Code einen Abschnitt zur Ausführung auf jeder Kachel in der Gruppe umfasst, der in jedem Abschnitt eine Instanz des Synchronisationsbefehls enthält.

Figurenliste

[0038] Als Hilfe zum Verständnis der vorliegenden Offenbarung und zur Veranschaulichung der Umsetzung von Ausführungsbeispielen wird beispielhaft auf die beigefügten Zeichnungsfiguren verwiesen, wobei:

Fig. 1 ein schematisches Blockschaltbild einer Multi-Thread-Verarbeitungseinheit zeigt,

Fig. 2 ein schematisches Blockschaltbild einer Vielzahl von Thread-Kontexten zeigt,

Fig. 3 ein Schema von verschachtelten Ausführungszeitschlitzen schematisch darstellt,

Fig. 4 einen Supervisor-Thread und eine Vielzahl von Worker-Threads schematisch darstellt,

Fig. 5 ein schematisches Diagramm der Logik zum Zusammenfassen von EXIT-Zuständen mehrerer Threads zeigt,

Fig. 6 eine Synchronisation zwischen Worker-Threads auf derselben Kachel schematisch darstellt,

Fig. 7 ein schematisches Blockschaltbild eines Prozessorchips mit mehreren Kacheln zeigt,

Fig. 8 ein Berechnungsmodell für eine massensynchrone Parallelverarbeitung (BSP) schematisch darstellt,

Fig. 9 eine weitere schematische Darstellung eines BSP-Modells zeigt,

Fig. 10 eine schematische Darstellung des BSP zwischen Multi-Thread-Verarbeitungseinheiten zeigt,

Fig. 11 ein schematisches Blockschaltbild eines Zwischenverbindungssystems zeigt,

Fig. 12 eine schematische Darstellung eines Systems mit mehreren zwischenverbundenen Prozessorchips zeigt,

Fig. 13 eine schematische Darstellung eines Multi-Kachel-BSP-Schemas zeigt,

Fig. 14 eine weitere schematische Darstellung eines Systems mit mehreren Prozessorchips zeigt,

Fig. 15 eine schematische Darstellung eines in einem Maschinenintelligenzalgorithmus verwendeten Graphen zeigt, und

Fig. 16 eine beispielhafte Verdrahtung zur Synchronisation zwischen Chips darstellt.

[0039] Es folgt eine Beschreibung einer Prozessorarchitektur, die in ihrem Befehlssatz einen dedizierten Befehl aufweist zum Durchführen einer Barrierensynchronisation und zum zeitgleichen Zusammenfassen von Ausstiegszuständen mehrerer Threads innerhalb mehrerer Kacheln zu einem einzigen zusammengefassten Zustand in einem Ausstiegszustandsregister, wobei dieses Register für den zusammengefassten Ausstiegszustand in jeder Kachel vorhanden ist und für jede der zusammengefassten Kacheln dasselbe Ergebnis enthält. Zuerst wird jedoch ein beispielhafter Prozessor, in dem dies integriert sein kann, unter Bezugnahme auf die **Fig. 1** bis **Fig. 4** beschrieben.

[0040] **Fig. 1** zeigt ein Beispiel eines Prozessormoduls **4** in Übereinstimmung mit Ausführungsbeispielen der vorliegenden Offenbarung. Das Prozessormodul **4** kann beispielsweise eine Kachel auf einem Array solcher Prozessorkacheln auf demselben Chip sein, oder kann implementiert sein als ein eigenständiger Prozessor auf seinem eigenen Chip. Das Prozessormodul **4** umfasst eine Multi-Thread-Verarbeitungseinheit **10** in der Form einer Barrel-Thread-Verarbeitungseinheit, und einen lokalen Speicher **11** (d.h., auf derselben Kachel im Falle eines Multi-Kachel-Arrays, oder demselben Chip im Falle eines Chips mit einzelner Prozessor). Bei einer Barrel-Thread-Verarbeitungseinheit handelt es sich um einen Typ von Multi-Thread-Verarbeitungseinheit, bei der die Ausführungszeit der Pipeline aufgeteilt ist in eine wiederholte Sequenz von verschachtelten Zeitschlitzen, von denen jeder einem vorgegebenen Thread gehören kann. Dies wird gleich näher erläutert. Der Speicher **11** umfasst einen Befehlsspeicher **12** und einen Datenspeicher **22** (die in verschiedenen adressierbaren Speichereinheiten oder verschiedenen Regionen derselben adressierbaren Speichereinheit implementiert sein können). Der Befehlsspeicher **12** speichert Maschinencode, der von der Verarbeitungseinheit **10** ausgeführt wird, während der Datenspeicher **22** sowohl von dem ausgeführten Code zu bearbeitende Daten als auch durch den ausgeführten Code auszugebende Daten (z.B. als Ergebnis solcher Operationen) speichert.

[0041] Der Speicher **12** speichert eine Vielzahl von verschiedenen Threads eines Programms, wobei jeder Thread eine jeweilige Sequenz von Befehlen zum Ausführen einer bestimmten Aufgabe oder bestimmten Aufgaben umfasst. Es gilt zu beachten, dass ein hier bezeichneter Befehl die Bedeutung eines Maschinencodebefehls hat, d.h., ein Exemplar eines der fundamentalen Befehle des Befehlssatzes des Prozessors, der aus einem einzelnen Befehlscode (Opcode) und keinem oder mehreren Operanden besteht.

[0042] Das hier beschriebene Programm umfasst eine Vielzahl von Worker-Threads und ein Supervisor-Unterprogramm, das als ein oder mehrere Supervisor-Threads strukturiert sein kann. Diese werden gleich näher erläutert. In Ausführungsbeispielen kann jeder von einigen oder allen der Worker-Threads die Form eines jeweiligen „Codelets“ aufweisen. Ein Codelet ist ein bestimmter Typ von Thread, manchmal auch als „atomischer“ Thread bezeichnet. Er umfasst die gesamte Eingangsinformation, die er zum Ausführen vom Beginn des Threads an benötigt (von dem Zeitpunkt seines Aufrufs), d.h., er übernimmt nach seinem Aufruf keine Eingabe von irgendeinem anderen Teil des Programms oder vom Speicher. Des Weiteren verwendet kein anderer Teil des Programms eine Ausgabe (Ergebnis) des Threads, bevor er abgeschlossen (beendet) wurde. Solange bei ihm kein Fehler auftritt, ist seine Beendigung garantiert. Nebenbei bemerkt wird in einiger Literatur ein Codelet auch als zustandslos definiert, d.h., falls er zweifach ausgeführt wird, kann er von der ersten Ausführung keine Information übernehmen, wobei aber eine solche zusätzliche Definition hier nicht übernommen wird. Es gilt auch zu beachten, dass es sich nicht bei allen Worker-Threads um Codelets (atomisch) handeln muss, und stattdessen einige oder alle Worker-Threads in Ausführungsbeispielen möglicherweise miteinander kommunizieren können.

[0043] Innerhalb der Verarbeitungseinheit **10** können mehrere verschiedene Threads von dem Befehlsspeicher **12** über eine einzelne Ausführungspipeline **13** verschachtelt werden (obwohl typischerweise lediglich eine Untergruppe aller in dem Befehlsspeicher gespeicherten Threads zu einem gegebenen Zeitpunkt in dem Gesamtprogramm verschachtelt werden kann). Die Multi-Thread-Verarbeitungseinheit **10** umfasst: eine Vielzahl von Kontextregisterdateien **26**, von denen jede angeordnet ist zum Wiedergeben des Zustands (Kontexts) eines anderen jeweiligen der zeitgleich auszuführenden Threads; eine gemeinsam genutzte Ausführungspipeline **13**, die den gleichzeitig ausgeführten Threads gemeinsam ist; und ein Scheduler **24** zur Durchführung einer Ablaufplanung der gleichzeitigen Threads zur Ausführung durch die gemeinsam genutzte Pipeline in verschachtelter Weise, vorzugsweise in einer Round-Robin-Weise. Die Verarbeitungseinheit **10** ist mit einem gemeinsam genutzten Befehlsspeicher **12** verbunden, der der Vielzahl von Threads gemeinsam ist, und einem gemeinsam genutzten Datenspeicher **22**, der ebenfalls der Vielzahl von Threads gemeinsam ist.

[0044] Die Ausführungspipeline **13** umfasst eine Hole-Stufe **14**, eine Decodierstufe **16** und eine Ausführungsstufe **18** mit einer Ausführungseinheit, die arithmetische und logische Operationen, Adressenberechnungen,

Lade- und Speicheroperationen und andere Operationen durchführen kann, wie durch die Befehlssatzarchitektur definiert ist. Jede der Kontextregisterdateien **26** umfasst eine jeweilige Gruppe von Registern zum Darstellen des Programmzustands eines jeweiligen Threads.

[0045] Ein Beispiel für die eine jede der Kontextregisterdateien **26** zusammenstellenden Register ist schematisch in **Fig. 2** dargestellt. Jede der Kontextregisterdateien **26** umfasst ein entsprechendes oder mehrere Steuerregister **28** mit zumindest einem Programmzähler (**PC**) für das jeweilige Thread (zum Verfolgen der Befehlsadresse, an der der Thread gerade ausgeführt wird), und in Ausführungsbeispielen auch eine Gruppe aus einem oder mehreren Zustandsregistern (**SR**), in denen ein aktueller Status des jeweiligen Threads aufgezeichnet ist (bspw. ob er gerade abläuft oder aufgrund eines Fehlers). Jede der Kontextregisterdateien **26** umfasst auch eine entsprechende Gruppe von Operandenregistern (**OP**) **32** zum vorübergehenden Speichern von Operanden der durch das jeweilige Thread ausgeführten Befehle, d.h., von den durch die Befehlscodes der jeweiligen Befehle des Threads bei deren Ausführung definierten Operationen bearbeitete oder resultierende Werte. Es ist ersichtlich, dass jede der Kontextregisterdateien **26** optional ein entsprechendes oder mehrere andere Typen von Registern (nicht gezeigt) umfassen kann. Es gilt auch zu beachten, dass während der Begriff „Registerdatei“ manchmal verwendet wird in Bezug auf eine Gruppe von Registern in einem gemeinsamen Adressraum, muss dies nicht notwendigerweise in der vorliegenden Offenbarung der Fall sein, und jeder der Hardwarekontexte **26** (jede der Registergruppen **26**, die jeden Kontext repräsentieren) kann allgemeiner eine oder oder mehrere solche Registerdateien umfassen.

[0046] Wie später näher erläutert wird, weist die offenbarte Anordnung eine Arbeiterkontextregisterdatei **CX0...CX(M-1)** für jeden der Anzahl **M** von gleichzeitig ausführbaren Threads auf ($M=3$ in dem dargestellten Beispiel, was aber nicht einschränkend ist), und eine zusätzliche Supervisorkontextregisterdatei **CXS**. Die Arbeiterkontextregisterdateien sind reserviert zum Speichern der Kontexte von Worker-Threads, und die Supervisorkontextregisterdatei ist reserviert zum Speichern des Kontexts eines Supervisor-Threads. Es gilt zu beachten, dass der Supervisor-Kontext in Ausführungsbeispielen dadurch besonders ist, dass er eine andere Zahl von Registern aufweist als jeder der Arbeiterkontexte. Jeder der Arbeiterkontexte weist vorzugsweise dieselbe Zahl von Statusregistern und Operandenregistern auf. In Ausführungsbeispielen kann der Supervisor-Kontext weniger Operandenregister aufweisen als jeder der Arbeiterkontexte. Beispiele für Operandenregister, die der Arbeiterkontext aufweisen kann, der Supervisorkontext aber nicht enthält, sind: Fließpunktregister, Akkumulierregister und/oder Register für dedizierte Gewichtungen (zum Speichern von Gewichtungen eines neuronalen Netzwerks). In Ausführungsbeispielen kann der Supervisor auch eine andere Zahl von Statusregistern aufweisen. Des Weiteren kann in Ausführungsbeispielen die Befehlssatzarchitektur des Prozessormoduls **4** so konfiguriert sein, dass die Worker-Threads und Supervisor-Thread(s) einige verschiedene Befehlstypen ausführen, aber auch einige Befehlstypen gemeinsam nutzen.

[0047] Die Hole-Stufe **14** ist so angeschlossen, dass sie auszuführende Befehle unter Steuerung des Schedulers **24** von dem Befehlsspeicher **12** holt. Der Scheduler **24** ist ausgestaltet zum Steuern der Hole-Stufe **14** zum Holen eines Befehls von jedem einer Gruppe von gleichzeitig ausgeführten Threads aufeinander folgend in einer wiederholenden Sequenz von Zeitschlitzten, sodass die Ressourcen der Pipeline **13** in eine Vielzahl von zeitlich verschachtelten Zeitschlitzten aufgeteilt werden, wie gleich näher erläutert wird. Das Schema der Ablaufsteuerung kann beispielsweise Round-Robin oder gewichtetes Round-Robin sein. Ein anderer Begriff für einen auf diese Weise betriebenen Prozessor ist ein Barrel-Thread-Prozessor.

[0048] In einigen Ausführungsbeispielen kann der Scheduler **24** Zugriff auf eines der Statusregister **SR** eines jeden Threads haben, das angibt, ob der Thread pausiert, sodass der Scheduler **24** im Grunde die Hole-Stufe **14** so steuert, dass lediglich die Befehle derjenigen Threads geholt werden, die gerade aktiv sind. In Ausführungsbeispielen gehört jeder Zeitschlitz (und jede entsprechende Kontextregisterdatei) vorzugsweise immer einem Thread oder einem anderen, d.h., jeder Schlitz ist immer von einem Thread besetzt, und jeder Schlitz ist immer in der Sequenz des Schedulers **24** enthalten; obwohl der einen gegebenen Schlitz besetzende Thread zurzeit möglicherweise pausiert, wobei in diesem Falle das Holen des Befehls für den entsprechenden Thread übersprungen wird, wenn die Sequenz bei diesem Schlitz ankommt. Alternativ ist beispielsweise nicht ausgeschlossen, dass in alternativen, weniger bevorzugten Implementierungen einige Schlitze vorübergehend vakant und von der ablaufgeplanten Sequenz ausgeschlossen sein können. Wird Bezug genommen auf die Zahl von Zeitschlitzten, die die Ausführungseinheit verschachteln kann, oder dergleichen, so bezieht sich dies auf die maximale Zahl von Schlitzen, die die Ausführungseinheit gleichzeitig ausführen kann, d.h., die Zahl von zeitgleichen Schlitzen, die die Hardware der Ausführungseinheit unterstützt.

[0049] Die Hole-Stufe **14** hat Zugriff auf den Programmzähler (**PC**) eines jeden der Kontexte. Für jeden entsprechenden Thread holt die Hole-Stufe **14** den nächsten Befehl dieses Threads aus der nächsten Adresse

in dem Programmspeicher **12**, wie durch den Programmzähler angegeben ist. Der Programmzähler wird mit jedem Ausführungszyklus inkrementiert, sofern er nicht aufgrund eines Abzweigungsbefehls abzweigt. Die Hole-Stufe **14** leitet den geholten Befehl dann an die Decodierstufe **16** weiter, um decodiert zu werden, und die Decodierstufe **16** leitet eine Anzeige des decodierten Befehls dann an die Ausführungseinheit **18** weiter, zusammen mit den decodierten Adressen eines jeden in dem Befehl spezifiziert Operandenregisters **32**, um den Befehl auszuführen.

[0050] Die Ausführungseinheit **18** hat Zugriff auf die Operandenregister **32** und das Steuerregister **28**, die sie verwenden kann beim Ausführen des Befehls basierend auf den decodierten Registeradressen, wie beispielsweise im Falle eines arithmetischen Befehls (z.B. durch Addieren, Multiplizieren, Subtrahieren oder Dividieren der Werte in zwei Operandenregistern und Ausgeben des Ergebnisses zu einem anderen Operandenregister des jeweiligen Threads). Oder, falls der Befehl einen Speicherzugriff (Laden oder Speichern) definiert, lädt die Lade-/Speicherlogik der Ausführungseinheit **18** einen Wert aus dem Datenspeicher in ein Operandenregister des entsprechenden Threads oder speichert einen Wert aus einem Operandenregister des entsprechenden Threads in den Datenspeicher **22**, in Übereinstimmung mit dem Befehl. Oder, falls der Befehl eine Abzweigung oder eine Statusänderung definiert, ändert die Ausführungseinheit den Wert in dem Programmzähler PC oder einem der Statusregister SR entsprechend. Es gilt zu beachten, dass, während ein Befehl eines Threads gerade von der Ausführungseinheit **18** ausgeführt wird, ein Befehl von dem Thread in dem nächsten Zeitschlitz in der verschachtelten Sequenz gerade durch die Decodierstufe **16** decodiert werden kann; und/oder während ein Befehl gerade von der Decodierstufe **16** decodiert wird, kann der Befehl von dem Thread in dem nächsten Zeitschlitz nach diesem von der Hole-Stufe **14** geholt werden (obwohl im Allgemeinen der Umfang der Offenbarung nicht beschränkt ist auf einen Befehl pro Zeitschlitz, zum Beispiel könnte in alternativen Szenarien ein Stapel von zwei oder mehreren Befehlen von einem gegebenen Thread pro Zeitschlitz abgegeben werden). Somit wird durch die Verschachtelung in vorteilhafter Weise Latenz in der Pipeline **13** verborgen, in Übereinstimmung mit bekannten Barrel-Thread-Verarbeitungstechniken.

[0051] Ein Beispiel für das durch den Scheduler **24** implementierte Verschachtelungsschema ist in **Fig. 3** dargestellt. Hier werden die gleichzeitigen Threads gemäß einem Round-Robin-Schema verschachtelt, wodurch die Runde innerhalb jeder Runde des Schemas in eine Sequenz von Zeitschlitzen **S0**, **S1**, **S2**... aufgeteilt wird, jeder zum Ausführen eines entsprechenden Threads. Typischerweise weist jeder Schlitz die Länge eines Prozessorzyklus auf und die verschiedenen Schlitze weisen eine gleiche Größe auf, obwohl dies nicht notwendigerweise in allen möglichen Ausführungsbeispielen so sein muss, zum Beispiel ist auch ein gewichtetes Round-Robin-Schema möglich, wodurch einige Threads mehr Zyklen pro Ausführungsrunde erhalten als andere. Im Allgemeinen kann das Barrel-Threading entweder eine gleichmäßige Round-Robin- oder eine gewichtete Round-Robin-Ablaufplanung verwenden, wobei im letzteren Fall die Gewichtung fest oder adaptiv sein kann.

[0052] Wie auch immer sich die Sequenz pro Ausführungsrunde gestaltet, wird dieses Muster dann wiederholt, wobei jede Runde ein entsprechendes Exemplar eines jeden Zeitschlitzes umfasst. Es gilt daher zu beachten, dass ein hier erwähnter Zeitschlitz die Bedeutung eines zugewiesenen Platzes in der Sequenz hat, nicht eines bestimmten Exemplars des Zeitschlitzes in einer gegebenen Wiederholung der Sequenz. Mit anderen Worten teilt der Scheduler **24** die Ausführungszyklen der Pipeline **13** in eine Vielzahl von zeitlich verschachtelten (zeitgemultiplexten) Ausführungskanälen ein, wobei jeder ein Wiederauftreten eines entsprechenden Zeitschlitzes in einer wiederholten Sequenz von Zeitschlitzen umfasst. In dem dargestellten Ausführungsbeispiel sind vier Zeitschlitze vorhanden, wobei dies aber lediglich erläuternden Zwecken dient und andere Zahlen möglich sind. Zum Beispiel sind in einem bevorzugten Ausführungsbeispiel tatsächlich sechs Zeitschlitze vorhanden.

[0053] Unabhängig davon, in wie viele Zeitschlitze das Round-Robin-Schema aufgeteilt ist, umfasst die Verarbeitungseinheit **10** dann entsprechend der vorliegenden Offenbarung eine Kontextregisterdatei **26** mehr, als Zeitschlitze vorhanden sind, d.h., sie unterstützt einen Kontext mehr, als die Anzahl verschachtelter Zeitschlitze, die sie durch Barrel-Threading verarbeiten kann.

[0054] Dies ist beispielhaft in **Fig. 2** dargestellt: Falls vier Zeitschlitze **S0**...**S3** vorhanden sind, wie in **Fig. 3** gezeigt, gibt es fünf Kontextregisterdateien, die hier markiert sind durch **CX0**, **CX1**, **CX2**, **CX3** und **CXS**. D.h., obwohl es lediglich vier Ausführungszeitschlitze **S0**...**S3** in dem Barrel-Thread-Schema gibt und somit nur vier Threads gleichzeitig ausgeführt werden können, wird hier offenbart, dass eine fünfte Kontextregisterdatei **CXS** mit einem fünften Programmzähler (**PC**), einer fünften Gruppe von Operandenregistern **32** und in Ausführungsbeispielen auch eine fünfte Gruppe von einem oder mehreren Statusregistern (**SR**) hinzugefügt werden. Wie erwähnt, ist aber zu beachten, dass der Supervisor-Kontext in Ausführungsbeispielen von den anderen

CX0...3 abweichen kann, und der Supervisor-Thread einen anderen Satz von Befehlen zum Betreiben der Ausführungspipeline **13** unterstützen kann.

[0055] Jeder der ersten vier Kontexte **CX0...CX3** wird verwendet zum Repräsentieren des Zustands einer jeweiligen aus einer Vielzahl von „Worker-Threads“, die aktuell einem der vier Ausführungszeitschlitz **S0...S3** zugeordnet sind zum Durchführen einer beliebigen von dem Programmierer gewünschten anwenderspezifischen Berechnungsaufgabe (es gilt wiederum zu beachten, dass es sich dabei lediglich um eine Untergruppe der Gesamtzahl von „Worker-Threads“ des in dem Befehlsspeicher **12** gespeicherten Programms handeln kann). Der fünfte Kontext **CXS** ist jedoch für eine spezielle Funktion reserviert, um den Zustand eines „Supervisor-Threads“ (**SV**) zu repräsentieren, dessen Rolle darin besteht, die Ausführung der Worker-Threads zu koordinieren, zumindest im Sinne einer Zuweisung dahingehend, welcher der Worker-Threads **W** in welchem der Zeitschlitz **S0, S1, S2...** an welcher Stelle in dem gesamten Programm ausgeführt werden soll. Optional kann der Supervisor-Thread andere „Aufseher“- oder koordinierende Zuständigkeiten haben. Beispielsweise kann der Supervisor-Thread verantwortlich sein für die Durchführung von Barrierensynchronisationen zum Sicherstellen einer bestimmten Ausführungsreihenfolge. Falls beispielsweise einer oder mehrere zweite Threads von Daten abhängig sind, die von einem oder mehreren auf demselben Prozessormodul **4** ablaufenden ersten Threads ausgegeben werden sollen, kann der Supervisor eine Barrierensynchronisation durchführen, um sicherzustellen, dass keiner der zweiten Threads beginnt, bevor die ersten Threads beendet wurden. Und/oder, der Supervisor kann eine Barrierensynchronisation durchführen, um sicherzustellen, dass einer oder mehrere Threads auf dem Prozessormodul **4** nicht beginnen, bevor eine bestimmte externe Datenquelle, wie beispielsweise eine andere Kachel oder ein anderer Prozessorchip, die zum Verfügbarmachen dieser Daten erforderliche Verarbeitung beendet hat. Der Supervisor-Thread kann auch verwendet werden zum Durchführen anderer auf die mehreren Worker-Threads bezogenen Funktionalitäten. So kann der Supervisor-Thread beispielsweise verantwortlich sein für die externe Kommunikation von Daten zu dem Prozessormodul **4** (zum Empfangen externer Daten zur Verarbeitung durch einen oder mehrere der Threads, und/oder zum Senden von durch einen oder mehrere der Worker-Threads ausgegebenen Daten). Im Allgemeinen kann der Supervisor-Thread verwendet werden zum Bereitstellen einer beliebigen Art von Überwachungs- oder Koordinierungsfunktion, die vom Programmierer gewünscht ist. So kann der Supervisor beispielsweise als weiteres Beispiel einen Transfer zwischen dem lokalen Speicher **12** der Kachel und einer oder mehrerer Ressourcen in dem erweiterten System (außerhalb des Arrays **6**), wie beispielsweise eine Speicherplatte oder Netzwerkkarte, überwachen.

[0056] Es gilt natürlich zu beachten, dass es sich bei den vier Zeitschlitzen lediglich um ein Beispiel handelt und im Allgemeinen in anderen Ausführungsbeispielen andere Zahlen verwendet werden können, sodass, falls ein Maximum von M Zeitschlitzen $0 \dots M-1$ pro Runde vorhanden ist, das Prozessormodul **4** $M+1$ Kontexte **CX...CX(M-1)** & **CXS** umfasst, d.h., einen für jeden Worker-Thread, der zu einem gegebenen Zeitpunkt verschachtelt werden kann, und ein zusätzlicher Kontext für den Supervisor. So gibt es in einer beispielhaften Implementierung sechs Zeitschlitz und sieben Kontexte.

[0057] Bezugnehmend auf **Fig. 4** weist der Supervisor-Thread **SV** in dem Schema der verschachtelten Zeitschlitz keinen eigenen Zeitschlitz per se auf. Auch nicht die Arbeiter, da die Zuordnung von Schlitzen zu Worker-Threads flexibel definiert ist. Stattdessen weist jeder Zeitschlitz seine eigene dedizierte Kontextregisterdatei (**CX0 ... CX(M-1)**) zum Speichern eines Arbeiterkontexts auf, der von dem Arbeiter verwendet wird, wenn der Schlitz dem Arbeiter zugewiesen wird, der aber nicht verwendet wird, wenn der Schlitz dem Supervisor zugewiesen wird. Wird ein gegebener Schlitz dem Supervisor zugewiesen, so verwendet der Schlitz stattdessen die Kontextregisterdatei **CXS** des Supervisors. Es gilt zu beachten, dass der Supervisor immer Zugriff auf seinen eigenen Kontext hat und keine Arbeiter die Supervisor-Kontextregisterdatei **CXS** belegen können.

[0058] Der Supervisor-Thread **SV** hat die Fähigkeit, in jedem beliebigen und allen der Zeitschlitz **S0...S3** (oder allgemeiner **S0 ... SM-1**) ausgeführt zu werden. Der Scheduler **24** ist so ausgestaltet, dass er dann, wenn das Programm als Ganzes startet, durch Zuweisen des Supervisor-Threads zu allen Zeitschlitzen beginnt, d.h., dass der Supervisor **SV** somit zu Beginn in allen der **S0...S3** ausgeführt wird. Der Supervisor-Thread ist jedoch mit einem Mechanismus ausgestattet, zum vorübergehenden Freigeben eines jeden der Schlitze, in dem er ausgeführt wird, für einen entsprechenden der Worker-Threads an einem nachfolgenden Zeitpunkt (entweder direkt oder nach Durchführung einer oder mehrerer Supervisor-Aufgaben), zum Beispiel anfänglich Arbeiter **W0...W3** in dem in **Fig. 4** gezeigten Beispiel. Dies wird dadurch erreicht, dass der Supervisor-Thread einen Freigabebefehl ausführt, der hier beispielhaft „RUN“ genannt ist. In Ausführungsbeispielen benötigt dieser Befehl zwei Operanden: eine Adresse eines Worker-Threads in dem Befehlsspeicher **12** und eine Adresse einiger Daten für diesen Worker-Thread in dem Datenspeicher **22**:

RUN task_addr, data_addr

[0059] Bei den Worker-Threads handelt es sich um Codeabschnitte, die gleichzeitig ausgeführt werden können, wobei jeder eine oder mehrere entsprechende auszuführende Berechnungsaufgaben repräsentiert. Die Datenadresse kann einige von dem Worker-Thread zu bearbeitende Daten spezifizieren. Als Alternative kann der Freigabebefehl lediglich einen einzelnen Operanden zum Spezifizieren der Adresse des Worker-Threads benötigen, und die Datenadresse könnte in dem Code des Worker-Threads enthalten sein; oder der einzelne Operand könnte in einem anderen Beispiel auf eine Datenstruktur zeigen, die die Adressen des Worker-Threads und der Daten spezifiziert. Wie erwähnt, können zumindest einige der Arbeiter in Ausführungsbeispielen die Form von Codelets aufweisen, d.h., atomistische Einheiten eines gleichzeitig ausführbaren Codes. Alternativ oder zusätzlich müssen einige der Arbeiter keine Codelets sein und können stattdessen fähig sein, miteinander zu kommunizieren.

[0060] Der Freigabebefehl („RUN“) wirkt auf den Scheduler **24** zum Freigeben des aktuellen Zeitschlitzes, in dem der Befehl selbst ausgeführt wird, an den durch den Operanden spezifizierten Worker-Thread. Es gilt zu beachten, dass der Freigabebefehl impliziert, dass der Zeitschlitz, in dem dieser Befehl ausgeführt wird, freigegeben wird (implizit bedeutet im Kontext von Maschinencodebefehlen, dass er zu dieser Spezifizierung keinen Operanden braucht - dies ergibt sich implizit aus dem Befehlscode selbst). Somit handelt es sich bei dem abgegebenen Zeitschlitz um denjenigen Zeitschlitz, in dem der Supervisor den Freigabebefehl ausführt. Oder, mit anderen Worten, der Supervisor wird in demselben Raum ausgeführt, den er abgibt. Der Supervisor ordnet an „führe diesen Teil des Codes an diesem Ort aus“, und dann gehört der wiederkehrende Schlitz ab diesem Zeitpunkt (vorübergehend) dem relevanten Worker-Thread.

[0061] Der Supervisor-Thread **SV** führt eine ähnliche Operation in jedem von einem oder mehreren anderen der Zeitschlitz durch, um einige oder alle dieser Zeitschlitz an andere entsprechende der Worker-Threads **W0...W3** abzugeben (ausgewählt von einer größeren Gruppe **W0...wj** in dem Befehlsspeicher **12**). Sobald er dies für den letzten Schlitz getan hat, ist der Supervisor suspendiert (später wird er dann an derjenigen Stelle wieder aufnehmen, wo er abgebrochen hat, wenn einer der Schlitz von einem Arbeiter **W** zurückgegeben wird).

[0062] Der Supervisor-Thread **SV** kann somit verschiedene Worker-Threads, die jeweils eine oder mehrere Aufgaben ausführen, verschiedenen der verschachtelten Ausführungszeitschlitz **S0...S3** zuweisen. Wenn der Supervisor-Thread feststellt, dass es Zeit zum Ausführen eines Worker-Threads ist, so verwendet er den Freigabebefehl („RUN“), um diesem Arbeiter denjenigen Zeitschlitz zuzuweisen, in dem der RUN-Befehl ausgeführt wurde.

[0063] In einigen Ausführungsbeispielen umfasst der Befehlssatz auch eine Variante des RUN-Befehls, RUNALL („alle freigeben“). Dieser Befehl wird verwendet zum gemeinsamen Starten einer Gruppe aus mehr als einem Arbeiter, die alle denselben Code ausführen. In Ausführungsbeispielen führt dies zu einem Starten eines Arbeiters in jedem der Schlitz **S0...S3** der Verarbeitungseinheit (oder allgemeiner **S0...S(M-1)**).

[0064] Ferner kopiert der RUN und/oder RUNALL-Befehl bei seiner Ausführung in einigen Ausführungsbeispielen auch automatisch einen Status von einem oder mehreren der Supervisor-Statusregister **CXS(SR)** in ein oder mehrere entsprechende Statusregister des/der durch den RUN- oder RUNALL-Befehl gestarteten Worker-Threads. Der kopierte Status kann beispielsweise einen oder mehrere Modi umfassen, wie beispielsweise einen Fließkommarundungsmodus (z.B. Runden auf den am nächsten kommenden Wert oder Runden auf null) und/oder ein Überlaufmodus (z.B. Sättigen oder Verwenden eines separaten, die Unendlichkeit repräsentierenden Werts). Der kopierte Status oder Modus steuert dann den betreffenden Arbeiter, um in Übereinstimmung mit dem kopierten Status oder Modus betrieben zu werden. In Ausführungsbeispielen kann der Arbeiter dieses später in seinem eigenen Statusregister überschreiben (er kann aber den Status des Supervisors nicht ändern). In weiteren alternativen oder zusätzlichen Ausführungsbeispielen können die Arbeiter auswählen, ob sie einen Status von einem oder mehreren Statusregistern des Supervisors lesen wollen (und wiederum ihren eigenen Status später ändern wollen). Dies kann beispielsweise wieder bedeuten, dass ein Modus von dem Supervisor-Statusregister übernommen wird, wie beispielsweise ein Fließkommamodus oder ein Rundungsmodus. In Ausführungsbeispielen kann der Supervisor jedoch nicht jedes der Kontextregister **CX0...** der Arbeiter lesen.

[0065] Sobald sie gestartet wurden, schreitet jeder der aktuell zugewiesenen Worker-Threads **W0...W3** zum Ausführen einer oder mehrerer Berechnungsaufgaben, die in dem durch den jeweiligen Freigabebefehl spezifizierten Code definiert sind. Am Ende dieses Vorgangs übergibt der jeweilige Worker-Thread dann den Zeitschlitz, in dem er ausgeführt wird, an den Supervisor-Thread zurück. Dies wird erreicht durch Ausführen eines Ausstiegsbefehls („EXIT“).

[0066] Der EXIT-Befehl benötigt zumindest einen Operanden und vorzugsweise nur einen einzigen Operanden, `exit_state` (z.B. ein binärer Wert), zur Verwendung für jeden vom Programmierer gewünschten Zweck zum Angeben eines Zustands des jeweiligen Codelets bei dessen Beendigung (z.B. zum Angeben, ob eine bestimmte Bedingung erfüllt war):

EXIT `exit_state`

[0067] Der EXIT-Befehl wirkt auf den Scheduler **24** in der Weise, dass der Zeitschlitz, in dem er ausgeführt wird, an den Supervisor-Thread zurückgegeben wird. Der Supervisor-Thread kann dann eine oder mehrere nachfolgende Supervisor-Aufgaben (z.B. Barrierensynchronisation und/oder Austausch von Daten mit externen Ressourcen wie beispielsweise anderen Kacheln), und/oder Fortführen der Ausführung eines weiteren Freigabebefehls, um dem betreffenden Schlitz einem neuen Worker-Thread (**W4**, usw.) zuzuweisen. Es gilt daher wiederum zu beachten, dass die Gesamtzahl der Threads in dem Befehlsspeicher **12** größer sein kann als die Zahl, die die Barrel-Thread-Verarbeitungseinheit **10** zu jedem Zeitpunkt verschachteln kann. Es ist Aufgabe des Supervisor-Threads **SV** zu steuern, welche der Worker-Threads **W0...Wj** aus dem Befehlsspeicher **12** an welcher Stufe in dem Gesamtprogramm welchem der verschachtelten Zeitschlitze **S0...SM** in der Round-Robin-Ablaufsteuerung des Schedulers **24** zugeteilt werden soll.

[0068] Ferner wird dem EXIT-Befehl eine weitere spezielle Funktion zugeteilt, nämlich zu bewirken, dass der in dem Operanden des EXIT-Befehls spezifizierte Ausstiegzzustand automatisch (durch eine dedizierte Hardwarelogik) mit den Ausstiegzzuständen einer Vielzahl von anderen Worker-Threads, die durch dieselbe Pipeline **13** desselben Prozessormoduls **4** (z.B. derselben Kachel) ausgeführt werden, vereinigt wird. Somit enthält der Befehl eine zusätzliche, implizite Möglichkeit zum Beenden eines Worker-Threads.

[0069] Eine Beispielschaltung für diesen Zweck ist in **Fig. 5** gezeigt. In diesem Beispiel weisen die Ausstiegzzustände der individuellen Threads und der zusammengefasste Ausstiegzzustand jeweils die Form eines einzelnen Bits auf, d.h., **0** oder **1**. Das Prozessormodul **4** umfasst ein Register **38** zum Speichern des zusammengefassten Ausstiegzzustands dieses Prozessormoduls **4**. Dieses Register kann hier als das „Lokalkonsens“-Register **\$LC** bezeichnet werden (im Gegensatz zu einem globalen Konsens, wenn das Prozessormodul **4** als eines aus einem Array von ähnlichen Prozessorkacheln enthalten ist, was gleich näher erläutert wird). In Ausführungsbeispielen ist das Lokalkonsensregister **\$LC 38** eines der Supervisor-Statusregister in der Supervisor-Kontextregisterdatei **CXS**. Die Logik zum Durchführen der Zusammenfassung umfasst ein UND-Gatter **37** ausgestaltet zum Durchführen einer logischen UND-Verknüpfung von (**A**) dem in dem Operanden des EXIT-Befehls spezifizierten Ausstiegzzustand und (**B**) dem aktuellen Wert in dem Lokalkonsensregister (**\$LC**) **38**, und zum Ausgeben des Ergebnisses (**Q**) zurück in das Lokalkonsensregister **\$LC 38** als neuer Wert der lokalen Zusammenfassung.

[0070] An einem geeigneten Synchronisationspunkt in dem Programm wird der in dem Lokalkonsensregister (**\$LC**) **38** gespeicherte Wert anfangs auf einen Wert **1** zurückgesetzt. D.h., jeder nach diesem Punkt aussteigende Thread wird bis zum nächsten Rücksetzen zu dem lokal zusammengefassten Ausstiegzzustand **\$LC** beitragen. Die Ausgabe (**Q**) des UND-Gatters **37** beträgt **1**, falls beide Eingaben (**A**, **B**) **1** sind, wobei aber andererseits die Ausgabe **Q** auf **0** wechselt, falls eine der Eingaben (**A**, **B**) **0** ist. Jedes Mal, wenn ein EXIT-Befehl ausgeführt wird, wird dessen Ausstiegzzustand mit den vorher aussteigenden (seit dem letzten Rücksetzen) zusammengefasst. Somit behält die Logik mittels der in **Fig. 5** gezeigten Anordnung eine laufende Zusammenfassung der Ausstiegzzustände aller Worker-Threads, die seit dem letzten Zurücksetzen des Lokalkonsensregisters (**\$LC**) **38** mittels eines EXIT-Befehls beendet wurden. In diesem Beispiel bezieht sich die laufende Zusammenfassung darauf, ob oder ob nicht alle bisherigen Threads mit wahren Zustand ausgestiegen sind: Jeder Ausstiegzzustand **0** eines jeden der Worker-Threads bedeutet, dass die Zusammenfassung in dem Register **38** auf **0** zwischengespeichert wird, bis zum nächsten Rücksetzen. In Ausführungsbeispielen kann der Supervisor **SV** die laufende Zusammenfassung zu jedem Zeitpunkt durch Zugreifen auf den aktuellen Wert des Lokalkonsensregisters (**\$LC**) **38** lesen (um dies zu tun, muss er nicht auf eine kachelinterne Synchronisation warten).

[0071] Das Rücksetzen der Zusammenfassung in dem Lokalkonsensregister (**\$LC**) **38** kann ausgeführt werden durch den Supervisor **SV** mittels eines **PUT** zu der Registeradresse des Lokalkonsensregisters (**\$LC**) **38** unter Verwendung eines oder mehrerer Allzweckbefehle, in diesem Beispiel zum Setzen eines Werts **1** in das Register **38**. Alternativ ist nicht ausgeschlossen, dass das Rücksetzen durch einen automatisierten Mechanismus erfolgen könnte, der beispielsweise durch Ausführen des hier später beschriebenen SYNC-Befehls getriggert wird.

[0072] Die Zusammenfassungsschaltung **37**, in diesem Falle das UND-Gatter, wird in einer dedizierten Hardware-Schaltung in der Ausführungseinheit der Ausführungsstufe **18** implementiert, unter Verwendung einer geeigneten Kombination von elektronischen Komponenten zur Bildung der Funktionalität einer Booleschen UND-Funktion. Eine dedizierte Schaltung oder Hardware bedeutet eine Schaltung mit festverdrahteter Funktion, im Gegensatz zu einer Programmierung in Software unter Verwendung eines Allzweckcodes. Die Aktualisierung des lokalen Ausstiegszustands wird getriggert durch die Ausführung des speziellen EXIT-Befehls, bei dem es sich um einen der fundamentalen Maschinencodebefehle in dem Befehlssatz des Prozessormoduls **4** handelt, der die inhärente Funktionalität des Zusammenfassens der Ausstiegszustände aufweist. Auch wird die lokale Zusammenfassung in einem Steuerregister **38** gespeichert, also einer dedizierten Speicherstelle (in Ausführungsbeispielen ein einzelnes Speicherbit), auf dessen Wert der in der Pipeline ablaufende Code zugreifen kann, das aber nicht von der Lade-Speicher-Einheit (**LSU**) nutzbar ist zum Speichern von Allzweckdaten. Stattdessen ist die Funktion der in einem Steuerregister gespeicherten Daten fest, in diesem Falle die Funktion des Speicherns des lokal zusammengefassten Ausstiegszustands. Vorzugsweise bildet das Lokalkonsensregister (**\$LC**) **38** eines der Steuerregister auf dem Prozessormodul **4** (z.B. einer Kachel), auf dessen Wert der Supervisor durch Ausführen eines GET-Befehls zugreifen kann und dessen Wert durch Ausführen eines PUT-Befehls gesetzt werden kann.

[0073] Es gilt zu beachten, dass die in **Fig. 5** gezeigte Schaltung lediglich ein Beispiel ist. Eine äquivalente Schaltung wäre das Ersetzen des UND-Gatters **37** durch ein ODER-Gatter und das Invertieren der Interpretation der Ausstiegszustände **0** und **1** in Software, d.h., $0 \rightarrow \text{wahr}$, $1 \rightarrow \text{falsch}$ (wobei das Register **38** bei jedem Synchronisationspunkt auf 0 statt 1 zurückgesetzt wird). In äquivalenter Weise, falls das UND-Gatter durch ein ODER-Gatter ersetzt wird, die Interpretation der Ausstiegszustände aber nicht invertiert wird, und auch nicht der Rücksetzwert, dann wird der zusammengefasste Zustand in **\$LC** aufzeichnen, ob einer (und nicht alle) der Arbeiterzustände mit dem Zustand **1** endete. In anderen Ausführungsbeispielen müssen die Ausstiegszustände nicht einzelnen Bits entsprechen. Zum Beispiel kann der Ausstiegszustand eines jeden individuellen Arbeiters ein einzelnes Bit sein, wobei aber der zusammengefasste Ausstiegszustand **\$LC** zwei Bits umfassen und einen trinären Zustand repräsentieren kann: Alle Arbeiter beendeten mit Zustand **1**, alle Arbeiter beendeten mit Zustand **0**, oder die Ausstiegszustände der Arbeiter waren gemischt. Als Beispiel für die Logik, um dies zu implementieren, kann eines der beiden den trinären Wert codierenden Bits ein Boolesches UND (oder ODER) der individuellen Ausstiegszustände sein, und das andere Bit des trinären Werts kann ein Boolesches ODER der individuellen Ausstiegszustände sein. Der dritte codierte Fall, der angibt, dass die Ausstiegszustände der Arbeiter gemischt waren, kann dann als das **XOR** dieser beiden Bits gebildet werden.

[0074] Die Ausstiegszustände können verwendet werden zum Darstellen, was immer der Programmierer wünscht, wobei aber ein speziell beabsichtigtes Beispiel darin besteht, einen Ausstiegszustand mit dem Wert **1** zu verwenden zum Anzeigen, dass der jeweilige Worker-Thread in einem „erfolgreichen“ oder „wahren“ Zustand beendet wurde, während ein Ausstiegszustand mit dem Wert **0** angibt, dass der jeweilige Worker-Thread in einem „nicht erfolgreichen“ oder „falschen“ Zustand beendet wurde (oder umgekehrt, falls die Zusammenfassungsschaltung **37** ein ODER anstelle eines UND durchführt und das Register **\$LC 38** anfänglich auf **0** zurückgesetzt wird). Es wird beispielsweise eine Anwendung angenommen, bei der jeder Worker-Thread eine Berechnung mit einer verknüpften Bedingung durchführt, wie beispielsweise eine Bedingung, die angibt, ob der/die Fehler in einem oder mehreren Parametern eines jeweiligen Knotens in dem Graphen eines Maschinenintelligenzalgorithmus innerhalb eines akzeptablen Niveaus gemäß einer vorbestimmten Metrik liegt/liegen. In diesem Fall kann ein individueller Ausstiegszustand mit einem logischen Pegel (z.B. 1) verwendet werden, um anzugeben, dass die Bedingung erfüllt ist (z.B. der Fehler oder die Fehler in dem einen oder mehreren Parameter des Knotens liegt/liegen innerhalb eines akzeptablen Niveaus gemäß einer Metrik); während ein individueller Ausstiegszustand mit entgegengesetztem logischen Pegel (z.B. 0) verwendet werden kann zum Anzeigen, dass die Bedingung nicht erfüllt war (z.B. der Fehler oder die Fehler liegen nicht innerhalb eines akzeptablen Niveaus gemäß der betreffenden Metrik). Bei der Bedingung kann es sich beispielsweise um eine für einen einzelnen Parameter oder jeden Parameter gesetzte Fehlerschwelle handeln, oder sie könnte eine komplexere Funktion aus einer Vielzahl von mit der entsprechenden von dem Worker-Thread ausgeführten Berechnung verknüpften Parametern sein.

[0075] Als ein weiteres komplexeres Beispiel können die individuellen Ausstiegszustände der Arbeiter und der zusammengefasste Ausstiegszustand jeweils zwei oder mehr Bits umfassen, was beispielsweise verwendet werden kann zum Darstellen eines Vertrauensgrads in die Ergebnisse der Worker-Threads. Zum Beispiel kann der Ausstiegszustand eines jeden individuellen Worker-Threads ein probabilistisches Maß an Vertrauen in ein Ergebnis des jeweiligen Worker-Threads darstellen, und die Zusammenfassungslogik **37** kann ersetzt werden durch eine komplexere Schaltung zum Durchführen einer probabilistischen Zusammenfassung der individuellen Vertrauensniveaus in Hardware.

[0076] Welche Bedeutung der Programmierer auch immer den Ausstiegszuständen beimisst, der Supervisor-Thread **SV** kann dann den zusammengefassten Wert aus dem Lokalkonsensregister (**\$LC**) **38** erhalten, um den zusammengefassten Ausstiegszustand aller Worker-Threads zu bestimmen, die seit dessen letzten Rücksetzen beendet wurden, beispielsweise an dem letzten Synchronisationspunkt, um beispielsweise zu bestimmen, ob alle Arbeiter in einem erfolgreichen oder wahren Zustand beendet haben. In Abhängigkeit dieses zusammengefassten Werts kann der Supervisor-Thread dann eine Entscheidung gemäß dem Design des Programmierers fällen. Der Programmierer kann wählen, welchen Nutzen er oder sie aus dem lokal zusammengefassten Ausstiegszustand ziehen möchte. Beispielsweise kann der Supervisor-Thread den lokal zusammengefassten Ausstiegszustand abfragen, um festzustellen, ob ein bestimmter Abschnitt des Programms, der aus einer bestimmten Untergruppe von Worker-Threads besteht, in erwarteter oder gewünschter Form beendet wurde. Falls nicht (z.B. hat zumindest einer der Worker-Threads in einem nicht erfolgreichen oder falschen Zustand geendet), kann er einem Host-Prozessor berichten, oder eine weitere Iteration des Programnteils mit denselben Worker-Threads durchführen; falls dem aber so ist (z.B. alle Worker-Threads endeten mit einem erfolgreichen oder wahren Zustand), kann er stattdessen zu einem anderen Teil des Programms mit einem oder mehreren neuen Arbeitern abzweigen.

[0077] Vorzugsweise sollte der Supervisor-Thread nicht auf den Wert in dem Lokalkonsensregister (**\$LC**) **38** zugreifen, bevor alle betreffenden Worker-Threads beendet wurden, sodass der darin gespeicherte Wert den korrekten aktuellen zusammengefassten Zustand aller gewünschten Threads repräsentiert. Das Warten darauf kann durchgesetzt werden durch eine von dem Supervisor-Thread ausgeführte Barriersynchronisation, um zu warten, bis alle aktuell laufenden lokalen Worker-Threads (d.h., diejenigen auf demselben Prozessor-Modul **4**, die durch dieselbe Pipeline **13** ablaufen) beendet sind. D.h., der Supervisor-Thread setzt das Lokalkonsensregister (**\$LC**) **38** zurück, startet eine Vielzahl von Worker-Threads, und initiiert dann eine lokale Barriersynchronisation (lokal in Bezug auf das Verarbeitungsmodul **4**, lokal in Bezug auf eine Kachel), um auf die Beendigung aller anhängigen Worker-Threads zu warten, bevor der Supervisor fortschreiten darf, um den zusammengefassten Ausstiegszustand aus dem Lokalkonsensregister (**\$LC**) **38** zu erhalten.

[0078] Bezugnehmend auf **Fig. 6** ist in Ausführungsbeispielen ein SYNC-Befehl (Synchronisationsbefehl) in dem Befehlssatz des Prozessors vorgesehen. Der SYNC-Befehl bewirkt, dass der Supervisor-Thread **SV** wartet, bis alle gerade ausgeführten Arbeiter **W** mittels eines EXIT-Befehls ausgestiegen sind. In Ausführungsbeispielen benötigt der SYNC-Befehl einen Modus als Operanden (in Ausführungsbeispielen kann es sich um seinen einzigen Operanden handeln), wobei der Modus spezifiziert, ob der SYNC-Befehl nur lokal in Bezug auf nur diejenigen Worker-Threads wirkt, die lokal auf demselben Prozessormodul **4**, zum Beispiel derselben Kachel, als Supervisor ausgeführt werden, als dessen Teil der SYNC-Befehl ausgeführt wird (d.h., nur Threads durch dieselbe Pipeline **13** derselben Barrel-Thread-Verarbeitungseinheit **10**); oder ob er stattdessen über mehrere Kacheln oder sogar über mehrere Chips angewendet wird.

SYNC - Modus // mode $\in$ {tile, chip, zone_1, zone_2}

[0079] Dies wird später näher erläutert, wobei aber für die Zwecke der **Fig. 6** ein lokaler SYNC-Befehl angenommen wird („SYNC-Kachel“, d.h., eine Synchronisation innerhalb einer einzelnen Kachel).

[0080] Die Arbeiter brauchen nicht als Operanden des SYNC-Befehls identifiziert zu werden, da der Supervisor **SV** dann implizit zum Warten veranlasst wird, bis keiner der Zeitschlitze **S0**, **S1**, ... der Barrel-Thread-Verarbeitungseinheit **10** von einem Arbeiter besetzt ist. Wie in **Fig. 6** gezeigt, führt der Supervisor einen SYNC-Befehl aus, sobald jeder eines aktuellen Stapels von Arbeitern **WLn** durch den Supervisor gestartet wurde. Falls der Supervisor **SV** in allen Schlitzen **S0...3** der Barrel-Thread-Verarbeitungseinheit **10** Arbeiter **W** startet (alle vier in dem dargestellten Beispiel, wobei es sich aber lediglich um eine Beispielimplementierung handelt), wird der SYNC-Befehl durch den Supervisor ausgeführt, sobald der erste des aktuellen Stapels von Worker-Threads **WLn** beendet ist, wodurch die Steuerung von zumindest einem Schlitz an den Supervisor **SV** zurückübergeben wird. Andererseits, falls die Arbeiter nicht alle Schlitze aufbrauchen, wird der SYNC-Befehl lediglich unmittelbar nach dem Starten des letzten Threads des aktuellen Stapels **WLn** ausgeführt. In beiden Fällen verursacht der SYNC-Befehl ein Warten des Supervisors **SV**, bis alle anderen des aktuellen Stapels von Arbeitern **WLn-1** einen EXIT-Befehl ausgeführt haben, bevor der Supervisor fortschreiten kann. Erst danach führt der Supervisor einen GET-Befehl aus, um den Inhalt des Lokalkonsensregisters (**\$LC**) **38** zu erhalten. Dieser Wartevorgang durch den Supervisor-Thread wird in Hardware umgesetzt, sobald der SYNC-Befehl ausgeführt wurde. D.h., im Ansprechen auf den Befehlscode des SYNC-Befehls veranlasst die Logik in der Ausführungseinheit (**EXU**) der Ausführungsstufe **18** ein Pausieren der Hole-Stufe **14** und des Schedulers **24** mit dem Ausgeben von Befehlen des Supervisor-Threads, bis alle anhängigen Worker-Threads einen EXIT-

Befehl ausgeführt haben. An einem Punkt nach dem Erhalten des Werts des Lokalkonsensregisters (**\$LC**) **38** (optional mit anderem Supervisor-Code dazwischen), führt der Supervisor einen PUT-Befehl aus, um das Lokalkonsensregister (**\$LC**) **38** zurückzusetzen (im dargestellten Beispiel auf **1**).

[0081] Wie ebenfalls in **Fig. 6** dargestellt ist, kann der SYNC-Befehl auch genutzt werden zum Platzieren von Synchronisationsbarrieren zwischen verschiedenen voneinander abhängigen Ebenen **WL1**, **WL2**, **WL3**, ... der Worker-Threads, wobei einer oder mehrere Threads in jeder aufeinander folgenden Ebene abhängig ist von Daten, die von einem oder mehreren Worker-Threads in seiner vorhergehenden Ebene ausgegeben werden. Der durch den Supervisor-Thread ausgeführte lokale SYNC-Befehl stellt sicher, dass keiner der Worker-Threads in der nächsten Ebene **WLn+1** ausgeführt wird, bevor alle Worker-Threads in der unmittelbar vorherigen Ebene **WLn** abgeschlossen sind (durch Ausführen eines EXIT-Befehls).

[0082] Wie erwähnt, kann das Prozessormodul **4** in Ausführungsbeispielen implementiert sein als eines aus einem Array von miteinander verbundenen Kacheln, die einen Multi-Kachel-Prozessor bilden, wobei jede Kachel ausgestaltet sein kann, wie oben in Bezug auf die **Fig. 1** bis **Fig. 6** beschrieben wurde.

[0083] Dies ist weitergehend dargestellt in **Fig. 7**, in der ein Einzelchipprozessor **2**, d.h., ein einzelnes Chip-Plättchen, gezeigt ist, mit einem Array **6** aus mehreren Prozessorkacheln **4** und einer chipinternen Zwischenverbindung **34**, die die Kacheln **4** verbindet. Der Chip **2** kann allein auf seiner eigenen integrierten Einzelchipschaltungspackung implementiert sein, oder als eines von vielen Chip-Plättchen, die in derselben IC-Packung gepackt sind. Die chipinterne Zwischenverbindung kann hier auch als „Austauschstruktur“ **34** bezeichnet werden, da sie den Kacheln ermöglicht, Daten untereinander auszutauschen. Jede Kachel **4** umfasst eine entsprechende Instanz der Barrel-Thread-Verarbeitungseinheit **10** mit Speicher **11**, die jeweils wie oben in Bezug auf die **Fig. 1** bis **Fig. 6** beschrieben, angeordnet sind. So kann beispielsweise der Chip **2** zur Veranschaulichung eine Größenordnung von Hunderten von Kacheln **4** oder sogar über tausend umfassen. Aus Vollständigkeitsgründen gilt es auch zu beachten, dass das hier bezeichnete „Array“ nicht notwendigerweise eine bestimmte Zahl von Dimensionen oder ein bestimmtes physikalisches Layout der Kacheln **4** impliziert.

[0084] In Ausführungsbeispielen umfasst jeder Chip **2** auch eine oder mehrere externe Verbindungen **8**, die ein Verbinden des Chips **2** mit einem oder mehreren anderen externen Prozessoren auf anderen Chips (z.B. eine oder mehrere andere Instanzen auf demselben Chip **2**) ermöglichen. Diese externen Verbindungen **8** können eine oder mehrere umfassen aus: eine oder mehrere Chip-zu-Host-Verbindungen zum Verbinden des Chips **2** mit einem Host-Prozessor, und/oder eine oder mehrere Chip-zu-Chip-Verbindungen zum Verbinden mit einer oder mehreren anderen Instanzen auf dem Chip **2** auf derselben IC-Packung oder Karte, oder auf verschiedenen Karten. In einer beispielhaften Anordnung empfängt der Chip **2** von einem Host-Prozessor (nicht gezeigt), der über eine der Chip-zu-Host-Verbindungen mit dem Chip verbunden ist, Arbeit in Form von Eingangsdaten, die durch den Chip **2** verarbeitet werden sollen. Mehrere Instanzen auf dem Chip **2** können durch Chip-zu-Chip-Verbindungen als Karten miteinander verbunden sein. Somit kann ein Host auf einen als Einzelchipprozessor **2** oder als mehrere möglicherweise auf mehreren miteinander verbundenen Karten angeordnete Einzelchipprozessoren **2** aufgebauten Computer zugreifen, abhängig von der für die Hostanwendung erforderlichen Arbeitslast.

[0085] Die Zwischenverbindung **34** ist ausgestaltet zum Ermöglichen einer Kommunikation zwischen den verschiedenen Prozessorkacheln **4** in dem Array **6** auf dem Chip **2**. Es können jedoch potenzielle Abhängigkeiten sowohl zwischen Threads auf derselben Kachel **4** als auch zwischen auf verschiedenen Kacheln **4** in dem Array **6** ablaufenden Programmteilen. Daher ist eine Technik erforderlich zum Verhindern, dass ein Codeabschnitt auf einer Kachel **4** vorausseilt gegenüber Daten, von denen es abhängig ist und die verfügbar gemacht werden durch einen anderen Codeabschnitt auf einer anderen Kachel **4**.

[0086] In Ausführungsbeispielen kann dies erreicht werden durch Implementieren eines Austauschschemas mit massensynchroner Parallelverarbeitung (**BSP**), wie in den **Fig. 8** und **Fig. 9** schematisch dargestellt ist.

[0087] Gemäß einer Version des BSP führt jede Kachel **4** in abwechselnden Zyklen eine Rechenphase **52** und eine Austauschphase **50** durch, die durch eine Barrierensynchronisation **30** zwischen Kacheln voneinander getrennt sind. In dem dargestellten Fall ist eine Barrierensynchronisation zwischen jeder Rechenphase **52** und der darauffolgenden Austauschphase **50** platziert. Während der Rechenphase **52** führt jede Kachel **4** eine oder mehrere Berechnungsaufgaben lokal auf der Kachel durch, kommuniziert aber keine Ergebnisse dieser Berechnungen mit irgendeiner anderen der Kacheln **4**. In der Austauschphase **50** darf jede Kachel **4** ein oder mehrere Ergebnisse der Berechnungen aus der vorherigen Rechenphase zu und/oder von einer oder mehreren anderen der Kacheln in der Gruppe austauschen, führt aber so lange keine neuen Berechnungen

durch, bis sie von anderen Kacheln **4** Daten empfangen hat, von denen ihre Aufgabe(n) abhängig ist/sind. Auch sendet sie keine Daten an irgendeine andere Kachel, mit Ausnahme derer, die in der vorherigen Rechenphase berechnet wurden. Es ist nicht ausgeschlossen, dass andere Operationen, wie beispielsweise interne steuerungsbezogene Operationen, in der Austauschphase durchgeführt werden. In Ausführungsbeispielen enthält die Austauschphase **50** keine nicht zeitdeterministischen Berechnungen, wobei aber eine geringe Zahl von zeitdeterministischen Berechnungen optional während der Austauschphase **50** erlaubt sein können. Es gilt auch zu beachten, dass eine Berechnungen durchführende Kachel **4** während der Rechenphase **52** mit anderen externen Systemressourcen außerhalb des Arrays von synchronisierten Kacheln **4** - zum Beispiel eine Netzwerkkarte, ein Plattenlaufwerk oder ein feldprogrammierbares Gate Array (FPGA) - möglicherweise kommunizieren darf, solange dies nicht eine Kommunikation mit anderen Kacheln **4** innerhalb der synchronisierten Gruppe involviert. Die Kommunikation außerhalb der Kachelgruppe kann optional den BSP-Mechanismus verwenden, sie kann aber alternativ die **BSP** nicht verwenden und stattdessen einen anderen, eignen Synchronisationsmechanismus verwenden.

[0088] Gemäß dem BSP-Prinzip wird eine Barrierensynchronisation **30** an der von den Rechenphasen **52** zu der Austauschphase **50** übergehenden Verbindungsstelle, oder an der von den Austauschphasen **50** zu der Rechenphase **52** übergehenden Verbindungsstelle, oder beiden platziert. D.h., entweder: (a) Alle Kacheln **4** müssen ihre jeweiligen Rechenphasen **52** abgeschlossen haben, bevor eine beliebige aus der Gruppe zur nächsten Austauschphase **50** fortschreiten darf, oder (b) alle Kacheln **4** in der Gruppe müssen ihre jeweiligen Austauschphasen **50** abgeschlossen haben, bevor irgendeine Kachel aus der Gruppe zur nächsten Rechenphase **52** fortschreiten darf, oder (c) beide dieser Bedingungen werden durchgesetzt. In allen drei Varianten sind es die individuellen Prozessoren, die zwischen Phasen wechseln, und die gesamte Anordnung, die synchronisiert wird. Die Sequenz aus Austausch- und Rechenphasen kann sich dann über mehrere Wiederholungen wiederholen. In der BSP-Terminologie wird jede Wiederholung einer Austauschphase und einer Rechenphase manchmal als „Superstep“ bezeichnet (obwohl es zu beachten gilt, dass die Terminologie in der Literatur nicht immer konsistent verwendet wird: manchmal wird jede individuelle Austauschphase und Rechenphase als Superstep bezeichnet, wogegen an anderer Stelle, wie bei der hier übernommenen Terminologie, die Austausch- und Rechenphasen gemeinsam als Superstep bezeichnet werden).

[0089] Es gilt ferner zu beachten, dass nicht ausgeschlossen ist, dass mehrere verschiedene unabhängige Gruppen von Kacheln **4** auf demselben Chip **2** oder verschiedenen Chips jeweils eine separate entsprechende BSP-Gruppe bilden können, die asynchron zueinander betrieben werden können, wobei der BSP-Zyklus des Berechnens, Synchronisierens und Austauschs lediglich innerhalb jeder gegebenen Gruppe aufgezwungen wird, wobei aber jede Gruppe dies unabhängig von den anderen Gruppen durchführt. D.h., ein Multi-Kachel-Array **6** kann mehrere intern synchronisierte Gruppen enthalten, die jeweils unabhängig von und asynchron zu den anderen Gruppen betrieben werden (später näher erläutert). In einigen Ausführungsbeispielen kann eine hierarchische Gruppierung der Synchronisation und des Austausches vorhanden sein, wie später näher erläutert wird.

[0090] Fig. 9 illustriert das BSP-Prinzip implementiert in einer Gruppe **4i**, **4ii**, **4iii** einiger oder aller Kacheln in dem Array **6** für einen Fall, der auferlegt: (a) eine Barrierensynchronisation von der Rechenphase **52** zu der Austauschphase **50** (siehe oben). Es gilt zu beachten, dass in dieser Anordnung einige Kacheln mit der Rechenphase **52** beginnen dürfen, während einige andere noch immer austauschen.

[0091] Gemäß den hier offenbarten Ausführungsbeispielen kann dieser Typ von **BSP** vereinfacht werden durch Aufnehmen einer zusätzlichen speziellen dedizierten Funktionalität in einen Maschinencodebefehl zum Durchführen einer Barrierensynchronisation, d.h. den SYNC-Befehl.

[0092] In Ausführungsbeispielen kann die SYNC-Funktion diese Funktionalität übernehmen, wenn sie durch eine kachelübergreifende Betriebsart als Operand qualifiziert ist, d.h., die chipinterne Betriebsart: SYNC Chip.

[0093] Dies ist schematisch in Fig. 10 dargestellt. Falls jede Kachel **4** eine Multi-Thread-Verarbeitungseinheit **10** umfasst, kann die Rechenphase **52** einer jeden Kachel tatsächlich Aufgaben umfassen, die durchgeführt werden durch mehrere Worker-Threads **W** auf derselben Kachel **4** (und eine vorgegebene Rechenphase **52** auf einer vorgegebenen Kachel **4** kann eine oder mehrere Ebenen **WL** von Worker-Threads umfassen, die im Falle mehrerer Ebenen durch interne Barrierensynchronisationen unter Verwendung des SYNC-Befehls mit der lokalen kachelinternen Betriebsart als Operand getrennt werden, wie zuvor beschrieben). Sobald der Supervisor-Thread **SV** auf einer vorgegebenen Kachel **4** den letzten Worker-Thread in dem aktuellen BSP-Superstep gestartet hat, dann führt der Supervisor auf dieser Kachel **4** einen SYNC-Befehl mit der kachelübergreifenden Betriebsart als Operand aus: SYNC Chip. Falls der Supervisor Worker-Threads in allen Schlitzen

seiner entsprechenden Verarbeitungseinheit **10** starten (RUN) muss, wird der „SYNC Chip“ ausgeführt, sobald der erste Schlitz, der nicht mehr für einen RUN weiterer Arbeiter in dem aktuellen BSP-Superstep benötigt wird, an den Supervisor zurückgegeben wird. Dies kann beispielsweise eintreten, nach dem EXIT des ersten Threads in der letzten Ebene WL, oder einfach nach dem EXIT des ersten Worker-Threads, falls nur eine einzelne Ebene vorhanden ist. Andererseits, falls nicht alle Schlitz für ablaufende Arbeiter in dem aktuellen BSP-Superstep verwendet werden müssen, kann der „SYNC Chip“ ausgeführt werden, sobald der letzte Arbeiter, der in dem aktuellen BSP-Superstep einen RUN durchführen muss, gestartet wurde. Dies kann auftreten, sobald alle Arbeiter in der letzten Ebene einem RUN unterzogen wurden, oder einfach sobald alle Worker-Threads einem RUN unterzogen wurden, falls lediglich eine Ebene vorhanden ist.

[0094] Die Ausführungseinheit (**EXU**) der Ausführungsstufe **18** ist so ausgestaltet, dass sie im Ansprechen auf den Befehlscode des SYNC-Befehls, falls dieser durch den chipinternen (kachelübergreifenden) Operanden qualifiziert ist, den Supervisor-Thread, in dem der „SYNC Chip“ ausgeführt wurde, zum Pausieren veranlasst, bis alle Kacheln **4** in dem Array **6** ihre ablaufenden Arbeiter beendet haben. Dies kann verwendet werden zum Implementieren einer Barriere zu dem nächsten BSP-Superstep. D.h., nachdem alle Kacheln **4** auf dem Chip **2** die Barriere passiert haben, kann das kachelübergreifende Programm als Ganzes zu der nächsten Austauschphase **50** fortschreiten.

[0095] **Fig. 11** zeigt ein schematisches Diagramm zur Darstellung der durch einen „SYNC Chip“ getriggerten Logik gemäß hier offenbarten Ausführungsbeispielen.

[0096] Sobald der Supervisor alle von ihm in der aktuellen Rechenphase **52** beabsichtigten Threads gestartet hat (**RUN**), führt er einen SYNC-Befehl mit dem chipinternen, kachelübergreifenden Operanden aus: SYNC Chip. Dadurch wird die nachfolgende in der dedizierten Synchronisationslogik **39** auf der Kachel **4** und in einer in der Hardwarezwischenverbindung **34** implementierten Synchronisationssteuerung **36** zu aktivierende Funktionalität getriggert. Diese Funktionalität sowohl der kachelinternen Synchronisationslogik **39** als auch der Synchronisationssteuerung **36** in der Zwischenverbindung **34** wird in einer dedizierten Hardwareschaltung so implementiert, dass, sobald der SYNC Chip ausgeführt ist, der Rest der Funktionalität ohne diesbezügliche Ausführung weiterer Befehle fortschreitet.

[0097] Als Erstes veranlasst die kachelinterne Synchronisationslogik **39** ein automatisches Pausieren der Befehlsausgabe für den Supervisor auf der fraglichen Kachel **4** (die Hole-Stufe **14** und der Scheduler werden veranlasst zum Unterdrücken der Ausgabe von Befehlen an den Supervisor). Sobald alle anhängigen Worker-Threads auf der lokalen Kachel **4** einen EXIT ausgeführt haben, sendet die Synchronisationslogik **39** automatisch eine Synchronisationsanforderung „**sync_req**“ an die Synchronisationssteuerung **36** in der Zwischenverbindung **34**. Die lokale Kachel **4** führt dann ihren Wartevorgang fort, wobei die Supervisor-Befehlsausgabe pausiert. Ein ähnlicher Prozess ist auch auf jeder der anderen Kacheln **4** in dem Array **6** implementiert (wobei jede ihre eigene Instanz der Synchronisationslogik **39** aufweist). Somit wird die Synchronisationssteuerung **36** an einem Punkt, an dem alle finalen Arbeiter in der aktuellen Rechenphase **52** auf allen Kacheln **4** in dem Array **6** beendet (EXIT) haben, eine entsprechende Synchronisationsanforderung (**sync_req**) von allen Kacheln **4** in dem Array **6** empfangen haben. Nur dann, im Ansprechen auf den Empfang des **sync_req** von jeder Kachel **4** in dem Array **6** auf demselben Chip **2** sendet die Synchronisationssteuerung **36** ein Synchronisationsbestätigungssignal „**sync_ack**“ an die Synchronisationslogik **39** auf jeder der Kacheln **4** zurück. Bis zu diesem Punkt hat jede der Kacheln **4** ihre Supervisor-Befehlsausgabe pausieren lassen und auf das Synchronisationsbestätigungssignal (**sync_ack**) gewartet. Beim Empfang des **sync_ack**-Signals beendet die Synchronisationslogik **39** auf der Kachel **4** automatisch das Pausieren der Supervisor-Befehlsausgabe für den entsprechenden Supervisor-Thread auf dieser Kachel **4**. Der Supervisor ist dann befreit zum Fortschreiten mit dem Austauschen von Daten mit anderen Kacheln **4** über die Zwischenverbindung **34** in einer nachfolgenden Austauschphase **50**.

[0098] Vorzugsweise werden die Signale **sync_req** und **sync_ack** zu bzw. von der Synchronisationssteuerung über eine oder mehrere dedizierte Synchronisationsleitungen, die jede Kachel **4** mit der Synchronisationssteuerung **36** in der Zwischenverbindung **34** verbindet, gesendet bzw. empfangen.

[0099] Des Weiteren ist gemäß den hier offenbarten Ausführungsbeispielen eine zusätzliche Funktionalität in dem SYNC-Befehl enthalten. D.h., zumindest dann, wenn er in einem kachelübergreifenden Modus (z.B. SYNC Chip) ausgeführt wird, veranlasst der SYNC-Befehl auch, dass die lokalen Ausstiegszustände **\$LC** einer jeden der synchronisierten Kacheln **4** automatisch in einer weiteren dedizierten Hardware **40** in der Zwischenverbindung **34** zusammengefasst werden. In den gezeigten Ausführungsbeispielen ist diese Logik durch ein UND-Gatter mit mehreren Eingängen (ein Eingang für jede Kachel **4** in dem Array **6**) gebildet, beispielsweise durch eine Kette aus UND-Gattern **40i**, **40ii**, ... mit zwei Eingängen, wie beispielhaft in **Fig. 11** gezeigt. Die-

se kachelübergreifende Zusammenfassungslogik **40** empfängt den Wert in dem lokalen Ausstiegzzustandsregister (Lokalkonsensregister) **\$LC 38** von jeder Kachel **4** in dem Array - in Ausführungsbeispielen jeweils ein einzelnes Bit - und fasst diese in einen einzelnen Wert zusammen, zum Beispiel eine UND-Verknüpfung aller lokal zusammengefasster Ausstiegzzustände. Somit bildet die Logik einen global zusammengefassten Ausstiegzzustand über alle Threads auf allen Kacheln **4** in dem Array **6**.

[0100] Jede der Kacheln **4** umfasst eine entsprechende Instanz eines Globalkonsensregisters (**\$GC**) **42** ausgestaltet zum Empfangen und Speichern des globalen Ausstiegzzustands von der globalen Zusammenfassungslogik **40** in der Zwischenverbindung **34**. In Ausführungsbeispielen handelt es sich dabei um ein weiteres Statusregister in der Kontextregisterdatei **CXS** des Supervisors. Im Ansprechen auf den Empfang der Synchronisationsanforderung (**sync_req**) von allen Kacheln **4** in dem Array **6** veranlasst die Synchronisationssteuerung **36**, dass die Ausgabe der Zusammenfassungslogik **40** (z.B. die Ausgabe des UND-Gatters) in dem Globalkonsensregister (**\$GC**) **42** auf jeder Kachel **4** gespeichert wird (es ist ersichtlich, dass der in **Fig. 11** gezeigte „Schalter“ eine schematische Darstellung dieser Funktionalität ist und die Aktualisierung tatsächlich durch eine beliebige geeignete digitale Logik implementiert wird). Auf dieses Register **\$GC 42** kann der Supervisor-Thread **SV** auf der entsprechenden Kachel **4** zugreifen, sobald die Supervisor-Befehlsausgabe wieder aufgenommen ist. In Ausführungsbeispielen ist das Globalkonsensregister **\$GC** als ein Steuerregister in der Steuerregisterdatei implementiert, sodass der Supervisor-Thread den Wert in dem Globalkonsensregister (**\$GC**) **42** mittels eines GET-Befehls erhalten kann. Es gilt zu beachten, dass die Synchronisationslogik **36** wartet, bis die **sync_req** von allen Kacheln **4** empfangen wurde, bevor sie den Wert in einem der Globalkonsensregister (**\$GC**) **42** aktualisiert, da andernfalls ein unrichtiger Wert für einen Supervisor-Thread auf einer Kachel verfügbar gemacht wird, der seinen Teil der Rechenphase **52** noch nicht abgeschlossen hat und daher noch immer abläuft.

[0101] Der global zusammengefasste Ausstiegzzustand **\$GC** ermöglicht dem Programm das Bestimmen eines Gesamtergebnisses von Programmteilen, die auf mehreren verschiedenen Kacheln **4** ablaufen, ohne den Zustand eines jeden individuellen Worker-Threads auf jeder individuellen Kachel individuell prüfen zu müssen. Er kann für jeden Zweck nach Wunsch des Programmierers verwendet werden. Beispielsweise in dem in **Fig. 11** gezeigten Beispiel, in dem die globale Zusammenfassung eine Boolesche UND-Funktion ist, bedeutet dies, dass jede Eingabe mit dem Wert **0** zu einer Zusammenfassung mit dem Wert **0** führt, wobei aber die Zusammenfassung den Wert **1** hat, wenn alle Eingaben den Wert **1** aufweisen. D.h., falls eine 1 zum Darstellen eines wahren oder erfolgreichen Ergebnisses verwendet wird, bedeutet dies, dass dann, wenn einer der lokalen Ausstiegzzustände einer beliebigen der Kacheln **4** falsch oder nicht erfolgreich ist, der global zusammengefasste Zustand ebenfalls falsch ist oder ein nicht erfolgreiches Ergebnis darstellt. Dies kann beispielsweise verwendet werden zum Bestimmen, ob oder nicht alle Teile des auf allen Kacheln ablaufenden Codes eine vorbestimmte Bedingung erfüllt haben. Somit kann das Programm ein einzelnes Register abfragen (in Ausführungsbeispielen ein einzelnes Bit), um zu fragen „Ging etwas schief? Ja oder Nein?“ oder „Haben alle Knoten in dem Graphen ein akzeptables Fehlerniveau erreicht? Ja oder Nein?“, statt die individuellen Zustände der individuellen Worker-Threads auf jeder individuellen Kachel prüfen zu müssen (und wiederum, in Ausführungsbeispielen ist es dem Supervisor tatsächlich nicht möglich, den Zustand der Arbeiter abzufragen, mit Ausnahme der Verwendung des Ausstiegzzustandsregisters **38, 42**). Mit anderen Worten reduzieren die Befehle **EXIT** und **SYNC** jeweils mehrere individuelle Ausstiegzzustände auf einen einzelnen kombinierten Zustand.

[0102] In einem beispielhaften Anwendungsfall kann der Supervisor auf einer oder mehreren der Kacheln einem Host-Prozessor berichten, falls die globale Zusammenfassung ein falsches oder nicht erfolgreiches Ergebnis anzeigte. Als weiteres Beispiel kann das Programm in Abhängigkeit des globalen Ausstiegzzustands eine Verzweigungsentscheidung treffen. Beispielsweise überprüft das Programm den global zusammengefassten Ausstiegzzustand **\$GC** und basierend darauf bestimmt es, ob eine Schleifenverarbeitung fortgeführt wird oder ob an eine andere Stelle abgezweigt wird. Falls der globale Ausstiegzzustand **\$GC** weiterhin falsch oder nicht erfolgreich ist, führt das Programm eine Iteration desselben ersten Teils des Programms fort, sobald aber der globale Ausstiegzzustand **\$GC** wahr oder erfolgreich ist, zweigt das Programm zu einem zweiten anderen Teil des Programms ab. Die Verzweigungsentscheidung kann individuell in jedem Supervisor-Thread implementiert werden, oder dadurch, dass einer der Supervisor eine Master-Rolle übernimmt und die anderen Slave-Supervisor auf den anderen Kacheln anweist (wobei die Master-Rolle in Software konfiguriert ist).

[0103] Es gilt zu beachten, dass die in **Fig. 11** gezeigte Zusammenfassungslogik **40** lediglich ein Beispiel darstellt. In einem anderen äquivalenten Beispiel kann das UND-Gatter durch ein ODER-Gatter ersetzt werden, und die Interpretation von **0** und **1** kann invertiert sein (**0** → wahr, **1** → falsch). Im äquivalenten Fall, falls das UND-Gatter durch ein ODER-Gatter ersetzt wird, die Interpretation der Ausstiegzzustände aber nicht invertiert ist, und auch nicht der Rücksetzwert, wird durch den zusammengefassten Zustand im **\$GC** aufgezeichnet, ob

eine (statt allen) der Kacheln mit einem lokal zusammengefassten Zustand **1** geendet hat. In einem anderen Beispiel kann der globale Ausstiegzzustand **\$GC** zwei Bits umfassen, die einen trinären Zustand darstellen: die lokal zusammengefassten Ausstiegzzustände **\$LC** aller Kacheln hatten den Zustand **1**, die lokal zusammengefassten Ausstiegzzustände **\$LC** aller Kacheln hatten den Zustand **0**, oder die lokal zusammengefassten Ausstiegzzustände **\$LC** der Kacheln waren gemischt. Als weiteres komplexeres Beispiel können die lokalen Ausstiegzzustände der Kacheln **4** und der global zusammengefasste Ausstiegzzustand zwei oder mehr Bits umfassen, was beispielsweise dazu verwendet werden kann, einen Konfidenzgrad in den Ergebnissen der Kacheln **4** darzustellen. Zum Beispiel kann der lokal zusammengefasste Ausstiegzzustand **\$LC** einer jeden individuellen Kachel ein statistisches, probabilistisches Maß der Konfidenz in ein Ergebnis der jeweiligen Kachel **4** sein, und die globale Zusammenfassungslogik **40** kann ersetzt werden durch eine komplexere Schaltung zum Durchführen einer statistischen Zusammenfassung der individuellen Konfidenzniveaus in Hardware.

[0104] Wie bereits erwähnt, können in Ausführungsbeispielen mehrere Instanzen des Chips **2** zusammengeschaltet werden, um ein noch größeres Array von Kacheln zu bilden, das sich über mehrere Chips **2** erstreckt. Dies ist in **Fig. 12** dargestellt. Einige oder alle der Chips **2** können auf derselben IC-Packung implementiert sein oder einige oder alle der Chips **2** können auf verschiedenen IC-Packungen implementiert sein. Die Chips **2** werden durch eine externe Zwischenverbindung **72** miteinander verbunden (über die in **Fig. 7** gezeigten externen Verbindungen **8**). Neben der Bereitstellung eines Kanals für den Austausch von Daten zwischen Kacheln **4** auf verschiedenen Chips stellt die externe Zwischenverbindung (Austauschperipherie) **72** auch eine Hardwareunterstützung zum Durchführen der Barriersynchronisation zwischen den Kacheln **4** auf verschiedenen Chips **2** und zum Zusammenfassen der lokalen Ausstiegzzustände der Kacheln **4** auf den verschiedenen Chips **2** bereit.

[0105] In Ausführungsbeispielen kann der SYNC-Befehl zumindest einen weiteren möglichen Wert für seinen Modus-Operanden benötigen, um eine externe, d.h., chipübergreifende, Synchronisation zu spezifizieren: SYNC zone_n, wobei zone_n eine externe Synchronisationszone repräsentiert. Die externe Zwischenverbindung **72** umfasst eine Hardwarelogik, die der in Bezug auf **Fig. 11** beschriebenen ähnelt, aber auf einem externen chipübergreifenden Maßstab. Wird der SYNC-Befehl mit einer in seinem Operanden spezifizierten externen Synchronisationszone von zwei oder mehreren Chips **2** ausgeführt, so führt dies zu einer Operation der Logik in der externen Zwischenverbindung **72** ähnlich der in Bezug auf die interne Zwischenverbindung **34** beschriebenen, aber die Kacheln **4** auf den mehreren verschiedenen Chips **2** in der spezifizierten Synchronisationszone übergreifend.

[0106] D.h., im Ansprechen auf einen externen SYNC-Befehl wird die Supervisor-Befehlsausgabe angehalten, bis alle Kacheln **4** auf allen Chips **2** in der externen Synchronisationszone ihre Rechenphase **52** abgeschlossen und eine Synchronisationsanforderung übermittelt haben. Ferner fasst eine Logik der externen Zwischenverbindung **72** die lokalen Ausstiegzzustände aller dieser Kacheln **4** über den mehreren Chips **2** in der fraglichen Zone zusammen. Sobald alle Kacheln **4** in der externen Synchronisationszone die Synchronisationsanforderung abgegeben haben, signalisiert die externe Zwischenverbindung **72** eine Synchronisationsbestätigung an die Kacheln **4** zurück und speichert den chipübergreifenden globalen zusammengefassten Ausstiegzzustand in dem Globalkonsensregister (**\$GC**) **42** für alle betreffenden Kacheln **4**. Im Ansprechen auf die Synchronisationsbestätigung nehmen die Kacheln **4** auf allen Chips **2** die Befehlsausgabe für den Supervisor wieder auf.

[0107] In Ausführungsbeispielen kann die Funktionalität der Zwischenverbindung **72** in den Chips **2** implementiert sein kann, d.h., die Logik kann unter den Chips **2** so aufgeteilt sein, dass lediglich verdrahtete Verbindungen zwischen den Chips erforderlich sind (die **Fig. 11** und **Fig. 12** sind schematisch).

[0108] Alle Kacheln **4** innerhalb der genannten Synchronisationszone sind so programmiert, dass dieselbe Synchronisationszone über den Modus-Operanden der jeweiligen SYNC-Befehle angezeigt wird. Falls dies aufgrund eines Programmierungsfehlers oder eines anderen Fehlers (wie beispielsweise ein Speicherparitätsfehler) nicht der Fall ist, ist die Synchronisationslogik **76** in der externen Zwischenverbindung **72** in Ausführungsbeispielen so ausgestaltet, dass einige oder alle Kacheln **4** keine Bestätigung empfangen werden, und das System somit an der nächsten externen Barriere zum Halten gebracht wird, sodass eine verwaltende externe CPU (z.B. dem Host) für ein Debug oder eine Systemwiederherstellung einschreiten kann. In anderen Ausführungsbeispielen wird ein Fehler angezeigt, falls die Synchronisationszonen nicht übereinstimmen. Vorzugsweise ist der Compiler aber so konfiguriert, dass alle Kacheln in derselben Zone zum relevanten Zeitpunkt dieselbe, richtige Synchronisationszone anzeigen.

[0109] **Fig. 13** zeigt einen beispielhaften BSP-Programmablauf, der sowohl eine interne (chipinterne) als auch eine externe (chipübergreifende) Synchronisation beinhaltet. Wie gezeigt, ist es bevorzugt, interne Austausch-

vorgänge **50** (von Daten zwischen Kacheln **4** auf demselben Chip **2**) von externen Austauschvorgänge **50'** (von Daten zwischen Kacheln **4** verschiedener Chips **2**) getrennt zu halten. Ein Grund dafür ist, dass ein durch den globalen SYNC-Befehl abgegrenzter globaler Austausch zwischen mehreren Chips hinsichtlich Latenz und Lastausgleichskomplexität „teurer“ sein kann, als lediglich eine chipinterne Synchronisation mit Austausch. Ein weiterer möglicher Grund liegt darin, dass der Austausch von Daten über die interne (chipseitige) Zwischenverbindung **34** zeitdeterministisch gestaltet werden kann, wogegen der Datenaustausch über die externe Zwischenverbindung **72** in Ausführungsbeispielen nicht zeitdeterministisch sein kann. In solchen Szenarien kann es nützlich sein, den internen und externen Austausch voneinander zu trennen, sodass der externe Synchronisations- und Austauschprozess nicht den internen Synchronisations- und Austauschprozess „kontaminiert“.

[0110] Dementsprechend kann das Programm in Ausführungsbeispielen zum Erzielen einer solchen Trennung ausgestaltet sein zum Durchführen einer Sequenz von Synchronisationen, Austauschphasen und Rechenphasen mit, in folgender Reihenfolge: (i) einer ersten Rechenphase, dann (ii) einer internen Barriersynchronisation **30**, dann (iii) einer internen Austauschphase **50**, dann (iv) einer externen Barriersynchronisation **80**, dann (v) einer externen Austauschphase **50'**. Vergleiche Chip **2II** in **Fig. 13**. Die externe Barriere **80** wird nach der internen Austauschphase **50** auferlegt, sodass das Programm erst nach dem internen Austausch **50** zu dem externen Austausch **50'** fortschreitet. Es gilt auch zu beachten, dass wie in Bezug auf Chip **2I** in **Fig. 12** gezeigt, eine Rechenphase optional zwischen dem internen Austausch (iii) und der externen Barriere (iv) enthalten sein kann. Die gesamte Sequenz wird durch das Programm erzwungen (das bspw. also solches mittels des Compilers erstellt wurde), und die interne Synchronisation mit Austausch erstreckt sich nicht auf Kacheln oder andere Einheiten aus einem anderen Chip **2**. Die Sequenz (i)-(v) (mit der vorgenannten optionalen Rechenphase zwischen (iii) und (iv)) kann in einer Reihe von Gesamtiterationen wiederholt werden. Pro Iteration können vor der externen Synchronisations- und Austauschphase mehrere Instanzen der internen Rechen-, Synchronisations- und Austauschphase (i)-(iii) vorhanden sein.

[0111] Es gilt zu beachten, dass die Kommunikationen während eines externen Austauschs **50** nicht ausschließlich auf externe beschränkt sind: einige Kacheln können lediglich interne Austauschvorgänge durchführen, einige können lediglich externe Austauschvorgänge durchführen, und einige können eine Mischung durchführen. Es gilt auch zu beachten, dass, wie in **Fig. 13** gezeigt, jeder beliebige Superstep eine Null-Rechenphase **52** oder eine Null-Austauschphase **50** aufweisen kann.

[0112] In einigen Ausführungsbeispielen können gemäß **Fig. 13** einige Kacheln **4** während einer Rechenphase eine lokale Ein-/Ausgabe durchführen, um bspw. Daten mit einem Host auszutauschen.

[0113] Wie in **Fig. 14** dargestellt ist, kann der Modus des SYNC-Befehls in Ausführungsbeispielen dazu verwendet werden, eine von mehreren verschiedenen möglichen externen Synchronisationszonen zu spezifizieren, zum Beispiel *zone_1* oder *zone_2*. In Ausführungsbeispielen können diese verschiedenen hierarchischen Ebenen entsprechen. D.h., jede höhere hierarchische Ebene **92** (z.B. Zone **2**) umfasst zwei oder mehr Zonen **91A**, **91B** aus zumindest einer niedrigeren hierarchischen Ebene. In Ausführungsbeispielen sind lediglich zwei hierarchische Ebenen vorhanden, wobei aber höhere Zahlen von verschachtelten Ebenen nicht ausgeschlossen sind. Falls der Operand des SYNC-Befehls auf die niedrigere hierarchische Ebene der externen Synchronisationszone (SYNC *zone_1*) gesetzt ist, werden die vorstehend beschriebenen Synchronisations- und Zusammenfassungsoperationen in Bezug auf die Kacheln **4** auf den Chips **2** lediglich in derselben externen Synchronisationszone auf niedrigerer Ebene, als die Kachel, auf der der SYNC-Befehl ausgeführt wurde, durchgeführt. Falls andererseits der Operand des SYNC-Befehls auf die höhere hierarchische Ebene der externen Synchronisationszone (SYNC *zone_2*) gesetzt ist, werden die vorstehend beschriebenen Synchronisations- und Zusammenfassungsoperationen automatisch in Bezug auf alle Kacheln **4** auf allen Chips **2** in der gleichen externen Synchronisationszone auf höherer Ebene durchgeführt, als die Kachel, auf der der SYNC-Befehl ausgeführt wurde. In Ausführungsbeispielen umfasst die höchste hierarchische Ebene der Synchronisationszonen alle Chips, d.h., sie wird verwendet zur Durchführung einer globalen Synchronisation. Werden mehrere Zonen niedrigerer Ebene verwendet, so kann **BSP** intern innerhalb der Gruppe von Kacheln **4** auf dem/den Chip(s) **2** innerhalb jeder Zone auferlegt werden, wobei aber jede Zone bzgl. der anderen Zonen asynchron betrieben werden kann, bis eine globale Synchronisation durchgeführt wird.

[0114] Es gilt zu beachten, dass in anderen Ausführungsbeispielen die durch den Modus des SYNC-Befehls spezifizierbaren Synchronisationszonen nicht auf eine hierarchische Natur beschränkt sind. Im Allgemeinen kann ein SYNC-Befehl mit Moden für jede Art von Gruppierung ausgestattet sein. Beispielsweise können die Moden eine Auswahl aus ausschließlich nicht hierarchischen Gruppen ermöglichen, oder einer Mischung aus hierarchischen Gruppen und einer oder mehrerer nicht hierarchischen Gruppen (wobei zumindest eine Gruppe nicht vollständig in der anderen Gruppe eingebettet ist). Dies ermöglicht in vorteilhafter Weise eine Flexibili-

tät für den Programmierer oder Compiler mit minimaler Codedichte zum Auswählen zwischen verschiedenen Layouts für intern-synchrone Gruppen, die gegenseitig asynchron sind.

[0115] Ein Beispielmechanismus zum Implementieren der Synchronisation zwischen der ausgewählten Synchronisationsgruppe **91**, **92** ist in **Fig. 16** dargestellt. Wie gezeigt, umfasst die externe Synchronisationslogik **76** in der externen Zwischenverbindung **72** einen entsprechenden Synchronisationsblock **95**, der mit jedem entsprechenden Chip verknüpft ist. Jeder Synchronisationsblock **95** umfasst eine entsprechende Torlogik und eine entsprechende Synchronisationszusammenfassung. Die Torlogik umfasst eine Hardwareschaltung, die die Chips **2** in einer Daisy-Chain-Topologie zum Zwecke der Synchronisation und Ausstiegzzustandszusammenfassung zusammenschaltet, und die Synchronisations- und Ausstiegzzustandsinformation wie folgt verbreitet. Der Synchronisations-Aggregator umfasst eine Hardwareschaltung ausgestaltet zum Zusammenfassen der Synchronisationsanforderungen (**sync_req**) und der Ausstiegzzustände wie folgt.

[0116] Der jeweilige mit jedem Chip **2** verknüpfte Synchronisationsblock **95** ist so mit seinem jeweiligen Chip **2** verbunden, dass er die durch diesen Chip **2** veranlasste Synchronisationsanforderung (**sync_req**) und den Ausstiegzzustand dieses Chips **2** erfassen kann und dass er die Synchronisationsbestätigung (**sync_ack**) und den globalen Ausstiegzzustand an den jeweiligen Chip **2** zurücksenden kann. Der jeweilige mit jedem Chip **2** verknüpfte Synchronisationsblock **95** ist auch mit dem Synchronisationsblock **95** von zumindest einem weiteren der Chips **2** über eine externe Synchronisationsschnittstelle mit einem Paket von vier Synchronisationsleitungen **96** verbunden, wobei Einzelheiten gleich näher erläutert werden. Dies kann ein Teil einer der Chip-zu-Chip-Verbindungen **8** sein. Im Falle einer Verbindung zwischen Chips auf verschiedenen Karten kann die Schnittstelle beispielsweise eine PCI-Schnittstelle umfassen und die vier Synchronisationsleitungen **96** können durch Wiederverwenden von vier Leitungen der PCI-Schnittstelle implementiert sein. Die Synchronisationsblöcke **95** einiger der Chips sind mit denen zweier benachbarter Chips verbunden, wobei jede Verbindung über entsprechende Instanzen der vier Synchronisationsleitungen **96** erfolgt. Auf diese Weise können die Chips **2** in einer oder mehreren Verkettungen (Daisy Chains) über ihre Synchronisationsblöcke **95** verbunden werden. Dies ermöglicht ein Ausbreiten der Synchronisationsanforderungen, Synchronisationsbestätigungen, Starten von Zusammenfassungen der Ausstiegzzustände und globalen Ausstiegzzustände entlang der Kette auf- und abwärts.

[0117] Im Betrieb wird der mit einem der Chips **2** verknüpfte Synchronisationsblock **95** für jede Synchronisationsgruppe **91**, **92** als Master für die Zwecke der Synchronisation und Ausstiegzzustandszusammenfassung festgelegt, wobei die übrigen in der Gruppe für diesen Zweck eine Slave-Funktion einnehmen. Jeder der Slave-Synchronisationsblöcke **95** wird mit der Richtung (z.B. links oder rechts) konfiguriert, in der er Synchronisationsanforderungen, Synchronisationsbestätigungen und Ausstiegzzustände für jede Synchronisationsgruppe **91**, **92** weitergeben soll (d.h. die Richtung zum Master). In Ausführungsbeispielen sind diese Einstellungen mittels Software konfigurierbar, zum Beispiel in einer anfänglichen Konfigurationsphase, nach der die Konfiguration während des nachfolgenden Betriebs des Systems eingestellt bleibt. Dieses kann beispielsweise durch den Host-Prozessor konfiguriert werden. Alternativ ist nicht ausgeschlossen, dass die Konfiguration auch festverdrahtet sein kann. In beiden Fällen können die verschiedenen Synchronisationsgruppen **91**, **92** verschiedene Master aufweisen und es ist im Allgemeinen für einen gegebenen Chip **2** (oder vielmehr seinen Synchronisationsblock **95**) möglich, Master einer Gruppe und nicht einer anderen Gruppe, der er angehört, zu sein, oder Master mehrerer Gruppen zu sein.

[0118] Zur Erläuterung wird beispielhaft das Beispielszenario in **Fig. 16** betrachtet. Es wird zum Beispiel angenommen, dass der Synchronisationsblock **95** des Chips **2IV** als der Master einer gegebenen Synchronisationsgruppe **91A** festgelegt ist. Nun wird der erste Chip **2I** in der Kette von Chips **2** betrachtet, die über ihre Synchronisationsblöcke **95** und Leitungen **96** bis hin zum Chip **2IV** verbunden sind. Wenn alle Worker-Threads der aktuellen Rechenphase auf dem ersten Chip **2I** einen EXIT-Befehl ausgeführt haben und die Supervisor auf allen (teilnehmenden) Kacheln **4** einen SYNC-Befehl, der die Synchronisationsgruppe **91A** spezifiziert, ausgeführt haben, dann signalisiert der erste Chip **2I** seine Synchronisationsbereitschaft zu seinem jeweils verknüpften Synchronisationsblock **95**. Der Chip **2I** gibt auch an seinen entsprechenden Synchronisationsblock **95** seinen zusammengefassten Ausstiegzzustand auf Chip-Ebene aus (die Zusammenfassung aller beendeten Arbeiter auf allen teilnehmenden Kacheln auf dem entsprechenden Chip **2I**). Im Ansprechen darauf leitet der Synchronisationsblock **95** des ersten Chips **2I** eine Synchronisationsanforderung (**sync_req**) an den Synchronisationsblock **95** des nächsten Chips **2II** in der Kette weiter. Ebenso leitet er den Ausstiegzzustand des ersten Chips **2I** an den Synchronisationsblock **95** dieses nächsten Chips **2II** weiter. Der Synchronisationsblock **95** dieses zweiten Chips **2II** wartet, bis alle Supervisor seiner eigenen (teilnehmenden) Kacheln **4** einen SYNC-Befehl, der die Synchronisationsgruppe **91A** spezifiziert, ausgeführt haben, was dazu führt, dass der zweite Chip **2II** seine Synchronisationsbereitschaft signalisiert. Erst dann leitet der Synchronisationsblock **95**

des zweiten Chips eine Synchronisationsanforderung an den Synchronisationsblock **95** des nächsten (dritten) Chips **2III** in der Kette weiter, und leitet auch eine laufende Zusammenfassung des Ausstiegszustands des ersten Chips **2I** mit der des zweiten Chips **2II** weiter. Falls der zweite Chip **2II** die Synchronisationsbereitschaft vor dem ersten Chip **2I** erreicht, dann müsste der Synchronisationsblock **95** des zweiten Chips **2II** auf die Synchronisationsanforderung des ersten Chips **2I** warten, bevor er die Synchronisationsanforderung zu dem Synchronisationsblock **95** des dritten Chips **2III** weiterleitet. Der Synchronisationsblock **95** des dritten Chips **2III** verhält sich in gleicher Weise, wobei er dieses Mal den laufenden zusammengefassten Ausstiegszustand von dem zweiten Chip **2II** zusammenfasst, um die nächste laufende Zusammenfassung zu erhalten und weiterzugeben, usw. Dies wird fortgeführt bis zum Master-Synchronisationsblock, in diesem Beispiel der auf Chip **2IV**.

[0119] Der Synchronisationsblock **95** des Masters bestimmt dann eine globale Zusammenfassung aller Ausstiegszustände basierend auf der von ihm empfangenen laufenden Zusammenfassung und des Ausstiegszustands seines eigenen Chips **2IV**. Er leitet diese globale Zusammenfassung gemeinsam mit der Synchronisationsbestätigung (**sync_ack**) entlang der Kette zu allen Chips **2** zurück.

[0120] Falls sich der Master auf einer Teilstrecke entlang einer Kette befindet, und nicht an einem Ende wie in dem vorstehend beschriebenen Beispiel, dann werden die Synchronisations- und Ausstiegszustandsinformationen in entgegengesetzten Richtungen auf jeder Seite des Masters ausgebreitet, auf beiden Seiten in Richtung des Masters. In diesem Falle gibt der Master die Synchronisationsbestätigung und den globalen Ausstiegszustand erst dann ab, wenn die Synchronisationsanforderung von beiden Seiten empfangen wurde. Es wird beispielsweise der Fall betrachtet, bei dem Chip **2III** Master der Gruppe **92** ist. Ferner könnte in Ausführungsbeispielen der Synchronisationsblock **95** von einigen der Chips **2** mit demjenigen von drei oder mehreren weiteren Chips **2** verbunden sein, sodass mehrere Zweige von Ketten in Richtung des Masters gebildet werden. Dann verhält sich jede Kette in der vorstehend beschriebenen Weise und der Master gibt die Synchronisationsbestätigung und den globalen Ausstiegszustand erst dann ab, wenn die Synchronisationsanforderung von allen Ketten empfangen wurde. Und/oder einer oder mehrere der Chips **2** könnten mit einer externen Quelle wie beispielsweise dem Host-Prozessor, einer Netzwerkkarte, einem Speichergerät oder einem FPGA verbunden sein.

[0121] In Ausführungsbeispielen wird die Signalisierung der Synchronisations- und Ausstiegszustandsinformation wie folgt implementiert. Das Bündel von vier Synchronisationsleitungen **96** zwischen jedem Paar von Chips **2** umfasst zwei Leitungspaare, ein erstes Paar **96_0** und ein zweites Paar **96_1**. Jedes Paar umfasst ein Exemplar einer Synchronisationsanforderungsleitung und ein Exemplar einer Synchronisationsbestätigungsleitung. Zur Signalisierung eines laufenden zusammengefassten Ausstiegszustands mit dem Wert **0** verwendet der Synchronisationsblock **95** des sendenden Chips **2** die Synchronisationsanforderungsleitung des ersten Leitungspaares **96_0** beim Signalisieren der Synchronisationsanforderung (**sync_req**), oder zum Signalisieren einer laufenden Zusammenfassung mit dem Wert **1** verwendet der Synchronisationsblock **95** die Synchronisationsanforderungsleitung des zweiten Leitungspaares **96_1** beim Signalisieren der Synchronisationsanforderung. Zum Signalisieren eines globalen zusammengefassten Ausstiegszustands des Werts **0** verwendet der Synchronisationsblock **95** des sendenden Chips **2** die Synchronisationsbestätigungsleitung des ersten Leitungspaares **96_0** beim Signalisieren der Synchronisationsbestätigung (**sync_ack**), oder zum Signalisieren einer globalen Zusammenfassung des Werts **1** verwendet der Synchronisationsblock **95** die Synchronisationsanforderungsleitung des zweiten Leitungspaares **96_1** beim Signalisieren der Synchronisationsbestätigung.

[0122] Es gilt zu beachten, dass es sich bei dem Obigen nur um den Mechanismus zum Weiterleiten der Synchronisations- und Ausstiegszustandsinformation handelt. Die aktuellen Daten (Inhalt) werden über einen anderen Kanal übertragen, wie beispielsweise später unter Bezugnahme auf **Fig. 16** erläutert wird. Ferner ist ersichtlich, dass es sich hierbei nur um eine beispielhafte Implementierung handelt und der Fachmann andere Schaltungen zur Implementierung der offenbarten Synchronisations- und Zusammenfassungsfunktionalität entwickeln kann, sobald ihm die hier offenbarte Spezifikation dieser Funktionalität vorliegt. Die Synchronisationslogik (**95** in **Fig. 16**) kann beispielsweise als Alternative zur dedizierten Verdrahtung Pakete verwenden, die über die Zwischenverbindung **34**, **72** übermittelt werden. Zum Beispiel können die **sync_req** und/oder die **sync_ack** jeweils als ein oder mehrere Pakete übertragen werden.

[0123] Die Funktionalität des SYNC-Befehls in den verschiedenen möglichen Betriebsarten wird nun zusammengefasst:

SYNC tile (führt eine lokale kachelinterne Barrierensynchronisation durch)

- Supervisor lässt Betriebsartübergänge ablaufen von der Ausführung bis zum Warten auf den Ausstieg der Arbeiter
- Befehlsausgabe für den Supervisor-Thread wird angehalten, bis alle Worker-Threads inaktiv sind
- Wenn alle Worker-Threads inaktiv sind, wird der zusammengefasste Arbeiterausstiegstatus über das Lokalkonsensregister (**\$LC**) **38** zugänglich gemacht.

SYNC chip (führt eine interne chipseitige Barrierensynchronisation durch)

- Supervisor lässt Betriebsartübergänge von der Ausführung bis zum Warten auf den Ausstieg der Arbeiter ablaufen
- Unterdrücken der Befehlsausgabe von dem Supervisor-Thread, bis alle Worker-Threads inaktiv sind
- Wenn alle Worker-Threads inaktiv sind:
 - der zusammengefasste lokale Arbeiterausstiegstatus wird über das Lokalkonsensregister (**\$LC**) **38** zugänglich gemacht
 - eine interne Synchronisationsteilnahme wird an die Austauschstruktur **34** signalisiert
 - der Supervisor bleibt inaktiv, bis die Kachel **4** eine interne Synchronisationsbestätigung von der Austauschstruktur **34** empfängt
 - der systemweite Ausstiegstatus wird in dem Globalkonsensregister (**\$GC**) **42** aktualisiert.

SYNC zone_n (führt eine externe Barrierensynchronisation innerhalb der Zone n durch)

- Supervisor lässt Betriebsartübergänge von der Ausführung bis zum Warten auf den Ausstieg der Arbeiter ablaufen
- Unterbrechen der Befehlsausgabe von dem Supervisor-Thread, bis alle Worker-Threads inaktiv sind.
- Wenn alle Worker-Threads inaktiv sind:
 - der zusammengefasste lokale Arbeiterausstiegstatus ist über das Lokalkonsensregister (**\$LC**) **38** verfügbar
 - eine externe Synchronisationsteilnahme wird dem externen System signalisiert, zum Beispiel der Synchronisationslogik in der vorgenannten externen Zwischenverbindung **72**
 - der Supervisor bleibt unterbrochen, bis die Kachel **4** eine externe Synchronisationsbestätigung von dem externen System **72** empfängt
 - der systemweite Ausstiegstatus wird in dem Globalkonsensregister (**\$GC**) **42** aktualisiert.

[0124] Wie bereits erwähnt, müssen nicht alle Kacheln **4** notwendigerweise an der Synchronisation teilnehmen. Wie erläutert, kann die Gruppe der teilnehmenden Kacheln in Ausführungsbeispielen durch den Modus-Operanden des Synchronisationsbefehls festgelegt werden. Dies ermöglicht jedoch nur eine Auswahl aus vordefinierten Gruppen von Kacheln. Es ist hier ersichtlich, dass es auch wünschenswert wäre, die Synchronisationsteilnahme kachelweise auszuwählen. Daher wird ein alternativer oder zusätzlicher Mechanismus zum Auswählen der an der Barrierensynchronisation teilnehmenden individuellen Kacheln **4** bereitgestellt.

[0125] Dies wird insbesondere dadurch erreicht, dass ein zusätzlicher Befehlstyp in dem Befehlssatz des Prozessors bereitgestellt wird, der anstelle des SYNC-Befehls durch eine oder einige Kacheln **4** ausgeführt wird. Dieser Befehl kann als der „Verzicht“-Befehl oder „SANS“-Befehl bezeichnet werden (starke automatische nicht teilnehmende Synchronisation). In Ausführungsbeispielen ist der SANS-Befehl für die Verwendung durch den Supervisor-Thread reserviert. In Ausführungsbeispielen benötigt er einen einzelnen unmittelbaren Operanden:

SANS n_barriers

[0126] Der SANS-Befehl bewirkt, dass die Kachel, auf der er ausgeführt wird, auf die aktuelle Barrierensynchronisation verzichtet, ohne aber die anderen Kacheln aufzuhalten, die auf den SYNC-Befehl aller Kacheln in der spezifizierten Synchronisationsgruppe warten.

[0127] Im Ergebnis sagt er „macht weiter ohne mich“. Wenn der SANS-Befehl ausgeführt wird, triggert der Befehlscode des SANS-Befehls die Logik in der Ausführungseinheit der Ausführungsstufe **18** zum Senden

einer Instanz des Synchronisationsanforderungssignals (**sync_req**) zu der internen und/oder externen Synchronisationssteuerung **36, 76** (abhängig von der Betriebsart). In Ausführungsbeispielen wird die durch den SANS-Befehl erzeugte Synchronisationsanforderung für jede Synchronisationsgruppe **91, 92** verwendet, die die Kachel **4**, die den SANS-Befehl ausgeführt hat, beinhaltet. D.h., unabhängig davon, welche Synchronisationsgruppe die Kacheln **4** in diesem lokalen Chip oder Chips als Nächstes verwenden (sie müssen die Synchronisationsgruppe vereinbaren), wird die Synchronisationsanforderung von denjenigen, die den SANS-Befehl ausgeführt haben, immer gültig bleiben.

[0128] Somit erscheint die den SANS-Befehl ausführende Kachel **4** aus der Perspektive der Synchronisationssteuerungslogik **36, 76** und der anderen Kacheln **4** in der Synchronisationsgruppe genau wie eine Kachel **4**, die einen SYNC-Befehl ausgeführt hat, und sie verzögert nicht die Synchronisationsbarriere und das Senden des Synchronisationsbestätigungssignals (**sync_ack**) von der Synchronisationslogik **36, 76**. D.h., die den SANS anstelle des SYNC-Befehls ausführenden Kacheln **4** bewirken weder eine Verzögerung noch eine Blockierung einer der anderen Kacheln **4**, die in einer beliebigen Synchronisationsgruppe involviert sind, der die betreffende Kachel andererseits angehört. Jeder durch einen SANS-Befehl ausgeführte Handshake ist für alle Synchronisationsgruppen **91, 92** gültig.

[0129] Entgegen des SYNC-Befehls führt der SANS-Befehl nicht zu einem Pausieren der Supervisor-Befehlsausgabe durch Warten auf das Synchronisationsbestätigungssignal (**sync_ack**) von der Synchronisationslogik **36, 76**. Stattdessen kann die entsprechende Kachel einfach weiterarbeiten, unbeschränkt durch die aktuelle Barrierensynchronisation, die zwischen den anderen Kacheln **4**, die SYNC-Befehle ausgeführt haben, durchgeführt wird. Somit ermöglicht der SANS-Befehl durch Nachahmen eines Synchronisationsbefehls ohne zu warten, dass seine Kachel **4** mit dem Bearbeiten einer oder mehrerer Aufgaben fortfahren kann, während die anderen Kacheln **4** noch immer synchronisieren können.

[0130] Der Operand **n_barriers** spezifiziert die Anzahl von „gebuchten“ Synchronisationen, d.h., die Anzahl von zukünftigen Synchronisationspunkten (Barrieren), bei denen die Kachel nicht beteiligt sein wird. Alternativ ist nicht ausgeschlossen, dass in anderen Ausführungsbeispielen der SANS-Befehl nicht diesen Operanden benötigt, und stattdessen jede Ausführung des SANS-Befehls lediglich einen einmaligen Verzicht verursacht.

[0131] Mittels des SANS-Befehls können bestimmte Kacheln **4** verantwortlich sein zum Ausführen von Aufgaben außerhalb des direkten Umfangs der BSP-Betriebsablaufsteuerung. Es kann beispielsweise wünschenswert sein, eine kleine Gruppe von Kacheln innerhalb eines Chips zuzuweisen zum Initiieren (und Bearbeiten) eines Datentransfers zu und/oder von einem Host-Speicher, während die Mehrheit der Kacheln **4** mit der/den primären Berechnungsaufgabe(n) beschäftigt sind. In solchen Szenarien können sich diejenigen Kacheln **4**, die nicht unmittelbar mit primärer Berechnung beschäftigt sind, selbst als für eine Zeitdauer effektiv von dem Synchronisationsmechanismus getrennt deklarieren, unter Verwendung des automatischen nicht teilnehmenden Synchronisationsmerkmals (SANS). Bei der Verwendung dieses Merkmals muss die Kachel **4** nicht aktiv (d.h., durch Ausführung des SYNC-Befehls) ihre Bereitschaft zur Synchronisation signalisieren (für eine beliebige der Synchronisationszonen), und liefert in Ausführungsbeispielen einen Null-Beitrag zu dem zusammengefassten Ausstiegszustand.

[0132] Der SANS-Befehl beginnt oder erstreckt sich über eine Periode, während der die Kachel **4**, auf der er ausgeführt wird, auf die aktive Teilnahme an der kachelübergreifenden Synchronisation verzichtet (oder eine Synchronisation mit anderen externen Ressourcen, falls diese ebenfalls in die Synchronisation einbezogen sind). Während dieser Periode wird diese Kachel **4** automatisch ihre Bereitschaft zur Synchronisation signalisieren, innerhalb aller Zonen, und in Ausführungsbeispielen auch einen Null-Beitrag zu dem globalen zusammengefassten Konsens **\$GC** liefern. Diese Zeitdauer kann als vorzeichenloser unmittelbarer Operand (**n_barriers**) ausgedrückt werden, der angibt, wie viele zusätzliche künftige Synchronisationspunkte automatisch durch diese Kachel **4** signalisiert werden. Bei der Ausführung des SANS-Befehls wird der durch seinen Operanden spezifizierte Wert **n_barriers** in ein Countdown-Register **\$ANS_DCOUNT** auf der entsprechenden Kachel **4** platziert. Dies ist ein Teil des architekturbezogenen Zustands, der verwendet wird zum Verfolgen, wie viele zusätzliche künftige Synchronisationsanforderungen erfolgen sollten. Falls der automatische nicht teilnehmende Synchronisationsmechanismus aktuell inaktiv ist, wird die erste Annahme einer Bereitschaft (Synchronisationsanforderung, **sync_req**) sofort durchgeführt. Nachfolgende Annahmen werden im Hintergrund stattfinden, sobald die vorherige Synchronisation fertiggestellt wurde (d.h., nach der Erklärung der Synchronisationsbestätigung, **sync_ack**). Falls der automatische nicht teilnehmende Synchronisationsmechanismus gerade aktiv ist, wird das Counter-Zähler-Register **\$ANS_DCOUNT** automatisch aktualisiert, sodass kein Synchronisationsbestätigungssignal unberücksichtigt bleibt. Der automatische nicht teilnehmende Synchronisationsmechanismus wird in dedizierter Hardwarelogik implementiert, vorzugsweise eine Instanz von diesem in jeder Kachel **4**, ob-

wohl in anderen Ausführungsbeispielen nicht ausgeschlossen ist, dass er stattdessen zentral für eine Gruppe von Kacheln oder alle Kacheln implementiert werden kann.

[0133] Bezüglich des Ausstiegzzustandsverhaltens gibt es tatsächlich eine Zahl von Möglichkeiten, abhängig von der Implementierung. In Ausführungsbeispielen fasst die Synchronisationslogik **36, 76** zum Erhalten des global zusammengefassten Ausstiegzzustands lediglich die lokalen Ausstiegzzustände derjenigen Kacheln **4** in der spezifizierten Synchronisationsgruppe zusammen, die einen SYNC-Befehl ausgeführt haben, und nicht derjenigen, die einen SANS-Befehl ausgeführt haben (die verzichtende Kachel oder Kacheln). Alternativ wird der globale zusammengefasste Ausstiegzzustand erhalten durch Zusammenfassen der lokalen Ausstiegzzustände aller Kacheln **4** in der Synchronisationsgruppe, die einen SYNC-Befehl ausgeführt hat, und derjenigen, die einen SANS-Befehl ausgeführt haben (sowohl die teilnehmenden als auch die verzichtenden Kacheln **4**). Im letzteren Fall kann der von der/den verzichtenden Kachel(n) **4** für die globale Zusammenfassung ausgegebene lokale Ausstiegzzustand der gerade lokal zusammengefasste Ausstiegzzustand der Arbeiter dieser Kachel zum Zeitpunkt der Ausführung des SANS-Befehls sein, genauso wie bei dem SYNC-Befehl (siehe Beschreibung des Lokalkonsensregisters **\$LC 38**). Alternativ kann der von der verzichtenden Kachel **4** ausgegebene lokale „Ausstiegzzustand“ ein voreingestellter Wert sein, beispielsweise der wahre Wert (z.B. logisch 1) in solchen Ausführungsbeispielen, bei denen der Ausstiegzzustand binär ist. Dadurch wird verhindert, dass die verzichtende Kachel **4** den globalen Ausstiegzzustand stört, in solchen Ausführungsbeispielen, bei denen jeder falsche lokale Ausstiegzzustand zu einem falschen globalen Ausstiegzzustand führt.

[0134] Was die Rückgabe des globalen Ausstiegzzustands angeht, gibt es zwei Möglichkeiten, unabhängig davon, ob die verzichtende Kachel einen lokalen Ausstiegzzustand zum Erzeugen der globalen Zusammenfassung zur Verfügung stellt oder nicht, und unabhängig davon, ob es sich bei dem Wert um einen aktuellen Wert oder einen voreingestellten Wert handelt. D.h., in einer Implementierung wird der durch die Synchronisationslogik **36, 76** in der Zwischenverbindung **34** erzeugte globale zusammengefasste Ausstiegzzustand lediglich in dem Globalkonsensregister **\$GC 42** derjenigen teilnehmenden Kachel **4** gespeichert, die einen SYNC-Befehl ausgeführt hat, und nicht in den verzichtenden Kacheln **4**, die stattdessen einen SANS-Befehl ausgeführt haben. In Ausführungsbeispielen wird stattdessen ein voreingestellter Wert in dem Globalkonsensregister **\$GC 42** derjenigen Kachel(n) **4** gespeichert, die einen SANS-Befehl ausgeführt haben (die verzichtenden Kacheln). Dieser voreingestellte Wert kann beispielsweise wahr sein, zum Beispiel logisch **1**, im Falle eines binären globalen Ausstiegzzustands. In einer alternativen Implementierung kann die durch die Synchronisationslogik **36, 76** erzeugte aktuelle globale Zusammenfassung jedoch in dem Globalkonsensregister **\$GC 42** sowohl der teilnehmenden Kacheln **4**, die SYNC-Befehle ausgeführt haben, als auch der verzichtenden Kacheln **4**, die stattdessen SANS-Befehle ausgeführt haben, gespeichert werden. Somit können alle Kacheln in der Gruppe noch immer Zugriff zu dem global zusammengefassten Ausstiegzzustand haben.

[0135] **Fig. 15** zeigt eine Beispielanwendung der hier offenbarten Prozessorarchitektur, nämlich eine Anwendung der Maschinenintelligenz.

[0136] Wie dem Fachmann auf dem Gebiet der Maschinenintelligenz bekannt ist, beginnt die Maschinenintelligenz mit einer Lernphase, in der der Maschinenintelligenzalgorithmus ein Wissensmodell lernt. Das Modell umfasst einen Graphen aus miteinander verbundenen Knoten (d.h. Knotenpunkten) **102** und Kanten (d.h. Verbindungen) **104**. Jeder Knoten **102** in dem Graphen weist eine oder mehrere Eingangskanten und eine oder mehrere Ausgangskanten auf. Einige der Eingangskanten von einigen der Knoten **102** sind die Ausgangskanten anderer Knoten, wodurch die Knoten zur Bildung des Graphen miteinander verbunden sind. Ferner bilden eine oder mehrere der Eingangskanten eines oder mehrerer der Knoten **102** die Eingänge des gesamten Graphen, und eine oder mehrere der Ausgangskanten eines oder mehrerer der Knoten **102** bilden die Ausgänge des gesamten Graphen. Manchmal kann ein gegebener Knoten sogar alles von diesem aufweisen: Eingänge in den Graphen, Ausgänge von dem Graphen und Verbindungen zu anderen Knoten. Jede Kante **104** kommuniziert einen Wert oder häufig einen Tensor (n-dimensionale Matrix), wobei diese die von und zu den Knoten **102** an ihren Eingangs- bzw. Ausgangskanten bereitgestellten Eingaben und Ausgaben bilden.

[0137] Jeder Knoten **102** repräsentiert eine Funktion aus seinen einen oder mehreren an seiner/seinen Eingangskante/-kanten empfangenen einen oder mehreren Eingaben, wobei das Ergebnis dieser Funktion der/den an der/den Ausgangskante oder-kanten bereitgestellten Ausgabe(n) entspricht. Jede Funktion ist parametrisiert durch einen oder mehrere jeweilige Parameter (manchmal als Gewichtungen bezeichnet, obwohl es sich nicht notwendigerweise um multiplikative Gewichte handeln muss). Im Allgemeinen können die durch die verschiedenen Knoten **102** repräsentierten Funktionen verschiedene Funktionsformen repräsentieren und/oder durch verschiedene Parameter parametrisiert sein.

[0138] Ferner ist jeder der einen oder mehreren Parameter einer jeden Knotenfunktion gekennzeichnet durch einen jeweiligen Fehlerwert. Darüber hinaus kann eine jeweilige Bedingung mit dem/den Fehler(n) in dem/den Parameter(n) eines jeden Knoten **102** verknüpft sein. Für einen Knoten **102**, der eine durch einen einzelnen Parameter parametrisierte Funktion repräsentiert, kann die Bedingung eine einfache Schwelle sein, d.h., die Bedingung ist erfüllt, falls sich der Fehler innerhalb der spezifizierten Schwelle befindet, nicht aber, falls der Fehler außerhalb der Schwelle liegt. Für einen durch mehr als einen jeweiligen Parameter parametrisierten Knoten **102** kann die Bedingung für das Erreichen eines akzeptablen Fehlerniveaus dieses Knoten **102** komplexer sein. Beispielsweise kann die Bedingung nur dann erfüllt sein, wenn jeder der Parameter dieses Knotens **102** unterhalb einer jeweiligen Schwelle liegt. Als weiteres Beispiel kann eine kombinierte Metrik definiert sein, die die Fehler in den verschiedenen Parametern desselben Knoten **102** kombiniert, und die Bedingung kann erfüllt sein, wenn der Wert der kombinierten Metrik unter einer spezifizierten Schwelle liegt, wogegen die Bedingung nicht erfüllt ist, falls der Wert der kombinierten Metrik außerhalb der Schwelle liegt (und umgekehrt in Abhängigkeit der Definition der Metrik). Unabhängig von der Bedingung wird hierdurch ein Maß bereitgestellt, ob der Fehler in dem/den Parameter(n) des Knoten unterhalb eines bestimmten Niveaus oder Grads an Akzeptanz liegt. Im Allgemeinen kann jede geeignete Metrik verwendet werden. Die Bedingung oder Metrik kann für alle Knoten gleich sein, oder für verschiedene der Knoten unterschiedlich.

[0139] In der Lernphase empfängt der Algorithmus Erfahrungsdaten, d.h., mehrere Datenpunkte, die verschiedene mögliche Kombinationen von Eingaben in den Graphen darstellen. Mit zunehmendem Empfang von Erfahrungsdaten stimmt der Algorithmus allmählich die Parameter der verschiedenen Knoten **102** in dem Graphen basierend auf den Erfahrungsdaten ab, mit dem Ziel einer Minimierung der Fehler in den Parametern. Ziel ist es, Parameterwerte aufzufinden, sodass die Ausgabe des Graphen so nahe wie möglich an einer gewünschten Ausgabe für eine gegebene Eingabe liegt. Wenn der Graph als Ganzes in die Nähe eines solchen Zustands gelangt, wird von einem Konvergieren des Graphen gesprochen. Nach einem geeigneten Grad des Konvergierens des Graphen kann dieser zur Durchführung von Vorhersagen oder Inferenzen verwendet werden, d.h., zum Vorhersagen eines Ergebnisses für eine gegebene Eingabe oder Herleiten einer Ursache für eine gegebene Ausgabe.

[0140] Die Lernphase kann eine Anzahl verschiedener möglicher Formen einnehmen. Beispielsweise bilden die eingegebenen Erfahrungsdaten bei einem überwachten Ansatz die Form von Trainingsdaten, d.h., Eingaben, die bekannten Ausgaben entsprechen. Mit jedem Datenpunkt kann der Algorithmus die Parameter so abstimmen, dass die Ausgabe besser mit der bekannten Ausgabe für die gegebene Eingabe übereinstimmt.

[0141] In der nachfolgenden Vorhersagephase kann der Graph dann dazu verwendet werden, eine Eingabe-anfrage mit einer näherungsweise vorhergesagten Ausgabe in Übereinstimmung zu bringen (oder umgekehrt, falls eine Inferenz durchgeführt wird). Andere Ansätze sind ebenfalls möglich. So gibt es beispielsweise bei einem unüberwachten Ansatz kein Konzept eines Referenzergebnisses pro eingegebenem Datenelement, und stattdessen wird es dem Maschinenintelligenzalgorithmus überlassen, seine eigene Struktur in den Ausgangsdaten zu identifizieren. Oder, in einem Verstärkungsansatz versucht der Algorithmus zumindest eine mögliche Ausgabe für jeden Datenpunkt in den eingegebenen Erfahrungsdaten, und es wird ihm mitgeteilt, ob diese Ausgabe positiv oder negativ ist (und eventuell einen Grad, mit dem sie positiv oder negativ ist), d.h., Erfolg oder Niederlage, oder Belohnung oder Bestrafung, oder dergleichen. Über viele Versuche kann der Algorithmus allmählich die Parameter des Graphen so abstimmen, dass Eingaben, die zu einem positiven Ergebnis führen, vorhergesagt werden können. Die verschiedenen Ansätze und Algorithmen zum Einlernen eines Graphen sind dem Fachmann auf dem Gebiet des Maschinenlernens bekannt.

[0142] Gemäß einer beispielhaften Anwendung der hier offenbarten Techniken wird jeder Worker-Thread zum Durchführen der mit einem jeweiligen der Knoten **102** in einem Maschinenintelligenzgraphen verknüpften Berechnungen programmiert. In diesem Falle entsprechen zumindest einige der Kanten **104** zwischen den Knoten **102** dem Datenaustausch zwischen Threads, und einige können Austauschvorgänge zwischen Kacheln involvieren. Des Weiteren werden die individuellen Ausstiegzzustände der Worker-Threads von dem Programmierer verwendet, um darzustellen, ob oder nicht der jeweilige Knoten **102** seine jeweilige Bedingung für die Konvergenz des/des Parametern) dieses Knotens erfüllt hat, d.h., liegt der Fehler in dem Parameter oder den Parametern innerhalb des akzeptablen Niveaus oder Bereichs im Fehlerraum. Beispielsweise handelt es sich dabei um eine beispielhafte Verwendung der Ausführungsbeispiele, bei denen jeder der individuellen Ausstiegzzustände ein individuelles Bit ist und der zusammengefasste Ausstiegzzustand einer UND-Verknüpfung der individuellen Ausstiegzzustände entspricht (oder äquivalent eine ODER-Verknüpfung, falls 0 als positiver Wert verwendet wird); oder bei denen der zusammengefasste Ausstiegzzustand ein trinärer Wert ist, der darstellt, ob die individuellen Ausstiegzzustände alle wahr, alle falsch oder gemischt sind. Somit kann das Pro-

gramm durch Prüfen eines einzelnen Registerwerts in dem Ausstiegswertregister **38** feststellen, ob der Graph als Ganzes oder zumindest ein Unterbereich des Graphen mit einem akzeptablen Grad konvergiert hat.

[0143] Als weitere Variante davon können Ausführungsbeispiele verwendet werden, bei denen die Zusammenfassung die Form einer statistischen Zusammenfassung individueller Konfidenzwerte einnimmt. In diesem Fall stellt jeder individuelle Ausstiegswert eine Konfidenz (z.B. einen Prozentsatz) dar, dass die Parameter des durch den entsprechenden Thread repräsentierten Knoten einen akzeptablen Fehlergrad erreicht haben. Der zusammengefasste Ausstiegswert kann dann verwendet werden zum Feststellen eines Gesamtgrads an Konfidenz dahingehend, ob der Graph, oder ein Unterbereich des Graphen, mit einem akzeptablen Grad konvergiert hat.

[0144] Im Falle einer Anordnung **6** mit mehreren Kacheln läuft in jeder Kachel ein Unterbereich des Graphen ab. Jeder Unterbereich umfasst ein Supervisor-Unterprogramm mit einem oder mehreren Supervisor-Threads, und eine Gruppe von Worker-Threads, wobei einer oder alle der Worker-Threads die Form von Codelets aufweisen können.

[0145] In solchen Anwendungen oder tatsächlich jeder auf einem Graphen basierenden Anwendung, in der jeder Worker-Thread zur Darstellung eines entsprechenden Knotens in einem Graphen verwendet wird, kann der von jedem Arbeiter umfasste „Codelet“ als eine Softwareprozedur definiert werden, die auf Grundlage des dauerhaften Zustands und der Eingaben und Ausgaben des einen Knotenpunkts betrieben wird, wobei der Codelet:

- auf Grundlage eines Worker-Thread-Registerkontexts gestartet wird, um in einem Barrel-Schlitz abzufließen, wenn der Supervisor-Thread einen „Run“-Befehl ausführt;
- bis zur Beendigung ausgeführt wird, ohne Kommunikation mit anderen Codelets oder dem Supervisor (mit Ausnahme der Rückkehr zum Supervisor, wenn der Codelet endet);
- auf den vorherrschenden Zustand eines Knotenpunkts mittels eines durch den „Run“-Befehl bereitgestellten Speicherzeigers Zugriff hat, und auf einen nicht dauerhaften Arbeitsbereich in einem Speicher, der diesem Barrel-Schlitz eigen ist; und
- einen „EXIT“ als seinen letzten Befehl ausführt, worauf der von ihm verwendete Barrel-Schlitz an den Supervisor zurückgegeben wird, und der durch den Ausstiegsbefehl spezifizierte Ausstiegswert mit dem für den Supervisor sichtbaren lokalen Ausstiegswert der Kachel zusammengefasst wird.

[0146] Das Aktualisieren eines Graphen (oder Untergraphen) bedeutet ein einmaliges Aktualisieren eines jeden konstituierenden Knotenpunkts in jeder mit der durch die Kanten definierten Kausalität konsistenten Reihenfolge. Das Aktualisieren eines Knotenpunkts bedeutet ein Ausführen eines Codelets auf Grundlage des Knotenpunktszustands. Ein Codelet ist eine Aktualisierungsprozedur für Knotenpunkte - ein Codelet ist üblicherweise mit vielen Knotenpunkten verknüpft. Der Supervisor führt einen RUN-Befehl pro Knotenpunkt aus, wobei jeder solche Befehl eine Knotenpunktszustandsadresse und eine Codeletadresse spezifiziert.

[0147] Es ist ersichtlich, dass die vorgenannten Ausführungsbeispiele lediglich beispielhaft beschrieben wurden.

[0148] Beispielsweise ist die Anwendbarkeit des Ausstiegswertzusammenfassungsmechanismus nicht auf die vorstehend beschriebene Architektur beschränkt, in der ein separater Kontext für den Supervisor-Thread vorgesehen ist, oder in der der Supervisor-Thread in einem Schlitz abläuft und er danach seinen Schlitz an einen Arbeiter abgibt. In einer anderen Anordnung kann der Supervisor beispielsweise in seinem eigenen dedizierten Schlitz ablaufen.

[0149] Ferner müssen die Begriffe „Supervisor“ und „Arbeiter“ nicht notwendigerweise bestimmte Verantwortlichkeiten implizieren, falls nicht anderweitig explizit behauptet ist, und insbesondere beschränken sie nicht als solche das vorstehend beschriebene Schema, bei dem ein Supervisor-Thread seinen Zeitschlitz an einen Arbeiter freigibt, usw. Im Allgemeinen kann ein Worker-Thread jeden Thread bezeichnen, dem eine Berechnungsaufgabe zugewiesen wird. Der Supervisor kann jeder Art von Überwachungs- oder Koordinierungs-Thread entsprechen, der verantwortlich ist für Aktionen wie: Zuweisen von Arbeitern zu Barrel-Schlitz, und/oder Durchführen der Barriersynchronisationen zwischen mehreren Threads, und/oder Durchführen einer Flusssteuerungsoperation (wie beispielsweise eine Abzweigung) in Abhängigkeit der Ergebnisse von mehr als einem Thread.

[0150] Wird auf eine Sequenz von verschachtelten Zeitschlitzten oder dergleichen Bezug genommen, so impliziert dies nicht notwendigerweise, dass die betreffende Sequenz alle möglichen oder verfügbaren Schlitzte umfasst. Beispielsweise könnte die fragliche Sequenz alle möglichen Schlitzte oder nur die gerade aktiven Schlitzte umfassen. Es ist nicht notwendigerweise ausgeschlossen, dass andere potenzielle Schlitzte vorhanden sind, die in der ablaufgeplanten Sequenz aktuell enthalten sind.

[0151] Der hier verwendete Begriff der Kachel ist nicht notwendigerweise beschränkend auf eine bestimmte Topografie oder dergleichen aufzufassen, und er kann im Allgemeinen auf jede modulare Einheit einer Verarbeitungsressource mit einer Verarbeitungseinheit **10** und einem entsprechenden Speicher **11** in einem Array von ähnlichen Modulen, typischerweise auf demselben Chip (demselben Chip-Plättchen) hinweisen.

[0152] Ferner ist der Umfang der vorliegenden Offenbarung nicht auf eine zeitdeterministische interne Zwischenverbindung und eine nicht zeitdeterministische externe Zwischenverbindung beschränkt. Der hier offenbarte Synchronisations- und Zusammenfassungsmechanismus kann auch in einer vollständig zeitdeterministischen Anordnung oder in einer vollständig nicht zeitdeterministischen Anordnung verwendet werden.

[0153] Während bestimmte Modi des SYNC-Befehls vorstehend beschrieben wurden, ist der Umfang der vorliegenden Offenbarung im Allgemeinen auch nicht auf solche Modi beschränkt. So ist beispielsweise die oben beschriebene Liste von Modi nicht notwendigerweise erschöpfend. Oder, in anderen Ausführungsbeispielen kann der SYNC-Befehl weniger Modi aufweisen, zum Beispiel muss der SYNC-Befehl nicht verschiedene hierarchische Ebenen der externen Synchronisation unterstützen, oder muss nicht zwischen chipinternen und chipübergreifenden Synchronisationen unterscheiden (d.h., in einem chipübergreifenden Modus agiert er immer in Bezug auf alle Kacheln, unabhängig davon, ob sie sich auf dem Chip oder außerhalb des Chips befinden). Gemäß noch weiteren alternativen Ausführungsbeispielen muss der SYNC-Befehl überhaupt keinen Modus mit einem Operanden aufweisen. Beispielsweise können in Ausführungsbeispielen getrennte Versionen des SYNC-Befehls (verschiedene Befehlscodes) für die verschiedenen Ebenen der Synchronisation und Ausstiegzzustandszusammenfassung vorgesehen sein (wie beispielsweise verschiedene SYNC-Befehle für die kachelinterne Synchronisation und kachelübergreifende chipinterne Synchronisation). Oder in anderen Ausführungsbeispielen kann ein dedizierter SYNC-Befehl lediglich für kachelübergreifende Synchronisationen vorgesehen sein (wobei die kachelinterne Synchronisation zwischen Threads ggf. in Allzwecksoftware durchgeführt wird).

[0154] Des Weiteren sind die Synchronisationszonen nicht darauf beschränkt, hierarchisch zu sein (d.h., eine innerhalb der anderen eingebettet), und in Ausführungsbeispielen können die wählbaren Synchronisationszonen aus einer oder mehreren nicht hierarchischen Gruppen bestehen oder solche enthalten (alle Kacheln dieser Gruppe sind nicht innerhalb einer einzelnen anderen wählbaren Gruppe eingebettet).

[0155] Ferner schließen die vorstehend beschriebenen Synchronisationsmöglichkeiten nicht die Involvierung anderer, von Multi-Kachel-Prozessoren verschiedener Ressourcen, zum Beispiel einen CPU-Prozessor, wie beispielsweise der Host-Prozessor, in Ausführungsbeispielen aus, und sogar nicht einen oder mehrere andere Komponenten, die keine Prozessoren sind, wie beispielsweise eine oder mehrere Netzwerkkarten, Speichergeräte und/oder FPGAs (zum Beispiel zum Kommunizieren globaler Synchronisationsnachrichten als Pakete über eine weitläufigere Zwischenverbindung anstelle der lediglich für Synchronisationszwecke verwendeten dedizierten Leitungen). So können beispielsweise einige Kacheln eine Teilnahme an Datentransfers mit einem externen System wählen, wobei diese Transfers die Berechnungslast für diese Kachel bilden. In diesem Falle sollten die Transfers vor der nächsten Barriere beendet sein. In einigen Fällen kann der Ausstiegzzustand der Kachel von einem Ergebnis der Kommunikation mit der externen Ressource abhängig sein, wobei diese Ressource den Ausstiegzzustand indirekt beeinflussen kann. Alternativ oder zusätzlich können andere Ressourcen als die Multi-Kachel-Prozessoren, beispielsweise der Host oder eine oder mehrere FPGAs, in dem Synchronisationsnetzwerk selbst integriert sein. D.h., ein Synchronisationssignal, wie beispielsweise **sync_req**, ist von dieser/diesen zusätzlichen Ressource(n) erforderlich, um die Barrierensynchronisation zu erfüllen und ein Weiterschreiten der Kacheln zur nächsten Austauschphase zu ermöglichen. Ferner kann der zusammengefasste globale Ausstiegzzustand in Ausführungsbeispielen einen Ausstiegzzustand der externen Ressource, zum Beispiel eines FPGAs, enthalten.

[0156] Andere Anwendungen und Varianten der offenbarten Techniken werden dem Fachmann unter Berücksichtigung der vorliegenden Offenbarung ersichtlich. Der Umfang der vorliegenden Offenbarung ist nicht beschränkt durch die beschriebenen Ausführungsbeispiele, sondern lediglich durch die beigefügten Ansprüche.

Patentansprüche

1. Verarbeitungssystem mit einer Anordnung von Kacheln und einer Zwischenverbindung zur Kommunikation zwischen den Kacheln, wobei:
jede Kachel eine Ausführungseinheit zum Ausführen von Maschinencodebefehlen aufweist, wobei jeder eine Instanz einer vordefinierten Gruppe von Befehlstypen in einem Befehlssatz des Prozessors ist, wobei jeder Befehlstyp in dem Befehlssatz durch einen entsprechenden Befehlscode und keine oder mehr Operandenfelder zur Aufnahme von keinem oder mehr Operanden definiert ist;
die Zwischenverbindung betreibbar ist zum Durchführen von Kommunikationen zwischen einer Gruppe von einigen oder allen der Kacheln gemäß einem massensynchronen Parallelschema, wodurch jede der Kacheln in der Gruppe eine kachelinterne Rechenphase gefolgt von einer kachelübergreifenden Austauschphase durchführt, wobei die Austauschphase zurückgehalten wird, bis alle Kacheln in der Gruppe die Rechenphase beendet haben, wobei jede Kachel in der Gruppe beim Beenden der Rechenphase einen lokalen Ausstiegzzustand aufweist;
der Befehlssatz einen Synchronisationsbefehl zur Ausführung durch jede Kachel in der Gruppe beim Beenden ihrer Rechenphase umfasst, wobei eine Ausführung des Synchronisationsbefehls die Ausführungseinheit zum Senden einer Synchronisationsanforderung an eine Hardwarelogik in der Zwischenverbindung veranlasst; und
die Logik in der Zwischenverbindung ausgestaltet ist zum Zusammenfassen der lokalen Ausstiegzzustände zu einem globalen Ausstiegzzustand, und zum Speichern des globalen Ausstiegzzustands in einem Globalausstiegzzustandsregister auf jeder der Kacheln in der Gruppe im Ansprechen auf eine Beendigung der Rechenphase durch alle Kacheln in der Gruppe, was durch einen Empfang der Synchronisationsanforderung von allen Kacheln in der Gruppe angezeigt ist, sodass der globale Ausstiegzzustand durch einen auf jeder der Kacheln in der Gruppe ablaufenden Codeabschnitt zugänglich ist.
2. Verarbeitungssystem nach Anspruch 1, wobei die Ausführungseinheit auf jeder Kachel ausgestaltet ist zum Pausieren einer Befehlsausgabe im Ansprechen auf eine Ausführung des Synchronisationsbefehls; und wobei die Logik in der Zwischenverbindung ausgestaltet ist zum Zurücksenden eines Synchronisationsbestätigungssignals zu jeder der Kacheln in der Gruppe im Ansprechen auf einen Empfang der Synchronisationsanforderung von allen Kacheln in der Gruppe, um die Befehlsausgabe wiederaufzunehmen.
3. Verarbeitungssystem nach einem der vorhergehenden Ansprüche, wobei jeder der lokalen Ausstiegzzustände und der globalen Ausstiegzzustände ein einzelnes Bit ist.
4. Verarbeitungseinheit nach Anspruch 3, wobei die Zusammenfassung aus einem Booleschen UND der lokalen Ausstiegzzustände oder einem Booleschen ODER der lokalen Ausstiegzzustände besteht.
5. Verarbeitungssystem nach Anspruch 1 oder 2, wobei der zusammengefasste Ausstiegzzustand zumindest zwei einen trinären Wert repräsentierende Bits umfasst, die angeben, ob die lokalen Ausstiegzzustände alle wahr, alle falsch oder gemischt sind.
6. Verarbeitungssystem nach einem vorhergehenden Anspruch, wobei jede der Gruppe von Kacheln ein Lokalausstiegzzustandsregister aufweist, das angeordnet ist zum Repräsentieren des lokalen Ausstiegzzustands der Kachel.
7. Verarbeitungssystem nach einem vorhergehenden Anspruch, wobei jede Kachel in der Gruppe umfasst: mehrere Kontextregistergruppen, wobei jede Kontextregistergruppe jeweils angeordnet ist zum Speichern eines Programmzustands eines jeweiligen der mehreren Threads; und einen Scheduler ausgestaltet zum Steuern des Ablaufs der Ausführung eines jeweiligen aus einer Vielzahl von Worker-Threads in jedem aus einer Vielzahl von Zeitschlitzen in einer wiederholten Sequenz verschachtelter Zeitschlitze, wobei der Programmzustand eines jeden der Worker-Threads in einem entsprechenden der Kontextregistergruppen gespeichert ist;
wobei die Austauschphase gemäß dem massensynchronen Parallelschema zurückgehalten ist, bis alle Worker-Threads auf allen Kacheln in der Gruppe die Rechenphase beendet haben;
wobei der lokale Ausstiegzzustand auf jeder Kachel eine Zusammenfassung eines von jedem der Worker-Threads auf der Kachel ausgegebenen individuellen Ausstiegzzustands ist; und
wobei der Codeabschnitt zumindest einen der mehreren Threads auf der Kachel umfasst.
8. Verarbeitungssystem nach Anspruch 7, wobei jede Kachel in der Gruppe eine Hardwarelogik umfasst, die ausgestaltet ist zum Durchführen der Zusammenfassung der individuellen Ausstiegzzustände zu dem lokalen Ausstiegzzustand.

9. Verarbeitungssystem nach Anspruch 8, wobei der Befehlssatz einen Ausstiegsbefehl zum Einschließen in jeden der Worker-Threads umfasst, wobei die Ausführungseinheit ausgestaltet ist zum Ausgeben des individuellen Ausstiegszustands des entsprechenden Worker-Threads und zum Beenden des entsprechenden Worker-Threads im Ansprechen auf den Befehlscode des Ausstiegsbefehls.

10. Verarbeitungssystem nach Anspruch 7, 8 oder 9, wobei jeder der individuellen Ausstiegszustände und der lokalen Ausstiegszustände ein einzelnes Bit ist, und die Zusammenfassung der individuellen Ausstiegszustände aus einem Booleschen UND der individuellen Ausstiegszustände oder einem Booleschen ODER der individuellen Ausstiegszustände besteht.

11. Verarbeitungssystem nach Anspruch 7, 8 oder 9, wobei der lokale Ausstiegszustand zumindest zwei einen trinären Wert repräsentierende Bits umfasst, der angibt, ob die individuellen Ausstiegszustände alle wahr, alle falsch oder gemischt sind.

12. Verarbeitungssystem nach einem der Ansprüche 7 bis 11, wobei die Austauschphase ausgestaltet ist zur Durchführung durch einen Supervisor-Thread getrennt von den Worker-Threads, und wobei der zumindest eine Thread den Supervisor-Thread umfasst.

13. Verarbeitungssystem nach Anspruch 12 bei dessen Abhängigkeit von Anspruch 2, wobei das Pausieren der Befehlsausgabe zumindest ein Pausieren der Befehlsausgabe von dem Supervisor-Thread bis zur Synchronisationsbestätigung umfasst.

14. Verarbeitungssystem nach Anspruch 12 oder 13, wobei die Kontextregistergruppen auf jeder Kachel umfassen: mehrere Arbeiterkontextregistergruppen angeordnet zum Repräsentieren des Programmzustands entsprechender der Vielzahl von Worker-Threads, und eine zusätzliche Supervisor-Kontextregistergruppe mit einer zusätzlichen Gruppe von Registern angeordnet zum Repräsentieren eines Programmzustands des Supervisor-Threads.

15. Verarbeitungssystem nach Anspruch 14, wobei:
der Supervisor-Thread ausgestaltet ist zum Starten durch Ablaufen in jedem der Zeitschlitze;
der Befehlssatz ferner einen Freigabebefehl umfasst und die Ausführungseinheit ausgestaltet ist zum Freigeben des Zeitschlitzes, in dem der Freigabebefehl ausgeführt wird, an den entsprechenden Worker-Thread im Ansprechen auf den Befehlscode des Freigabebefehls; und
der Ausstiegsbefehl eine Rückgabe des entsprechenden Zeitschlitzes, in dem der Ausstiegsbefehl ausgeführt wird, an den Supervisor-Thread veranlasst, sodass der Supervisor-Thread den Ablauf in dem entsprechenden Schlitz wiederaufnimmt.

16. Verarbeitungssystem nach einem vorhergehenden Anspruch, das mit dem Code programmiert ist; wobei der Codeabschnitt ausgestaltet ist zum Verwenden des globalen Ausstiegszustands, sobald er gültig ist, zum Durchführen einer von dem globalen Ausstiegszustand abhängigen Abzweigungsentscheidung.

17. Verarbeitungssystem nach einem vorhergehenden Anspruch, programmiert zum Durchführen eines Maschinenintelligenzalgorithmus, bei dem jeder Knoten in einem Graphen eine oder mehrere jeweilige Eingangskanten und eine oder mehrere jeweilige Ausgangskanten aufweist, wobei die Eingangskanten zumindest einiger der Knoten den Ausgangskanten zumindest einiger anderer der Knoten entsprechen, wobei jeder Knoten eine entsprechende seine Ausgangskanten mit seinen Eingangskanten in Beziehung bringende Funktion umfasst, wobei jede entsprechende Funktion durch einen oder mehrere entsprechende Parameter parametrisiert ist, und wobei jeder der entsprechenden Parameter einen zugehörigen Fehler aufweist, sodass der Graph mit zunehmender Verringerung der Fehler in einigen oder allen Parametern in Richtung einer Lösung konvergiert; wobei jede der Kacheln einen entsprechenden Untergraphen mit einer Untergruppe der Knoten in dem Graphen modelliert, und jeder der lokalen Ausstiegszustände verwendet ist zum Angeben, ob die Fehler in dem einen oder mehreren Parametern der Knoten in dem entsprechenden Untergraphen eine vorbestimmte Bedingung erfüllt haben.

18. Verarbeitungssystem nach einem vorhergehenden Anspruch, wobei die Gruppe zumindest teilweise durch einen Operanden des Synchronisationsbefehls ausgewählt ist.

19. Verarbeitungssystem nach Anspruch 18, wobei der Operand des Synchronisationsbefehls auswählt, ob ausschließlich Kacheln auf demselben Chip oder Kacheln auf verschiedenen Chips in der Gruppe enthalten sind.

20. Verarbeitungssystem nach Anspruch 18 oder 19, wobei der Operand des Synchronisationsbefehls die Gruppe aus verschiedenen hierarchischen Gruppierungsebenen auswählt.

21. Verarbeitungssystem nach einem vorhergehenden Anspruch, wobei der Befehlssatz ferner einen Verzichtsbefehl enthält, der die Kachel, auf der der Verzichtsbefehl ausgeführt wird, zur Nichtteilnahme an der Gruppe veranlasst.

22. Verfahren zum Betreiben eines Verarbeitungssystems mit einer Anordnung von Kacheln und einer Zwischenverbindung zur Kommunikation zwischen den Kacheln, wobei jede Kachel eine Ausführungseinheit zum Ausführen von Maschinencodebefehlen umfasst, wobei jeder eine Instanz einer vordefinierten Gruppe von Befehlstypen in einem Befehlssatz des Prozessors ist, wobei jeder Befehlstyp in dem Befehlssatz durch einen entsprechenden Befehlscode und keine oder mehr Operandenfelder zur Aufnahme von keinem oder mehr Operanden definiert ist; wobei das Verfahren umfasst:

Durchführen von Kommunikationen zwischen einer Gruppe aus einigen oder allen der Kacheln, über die Zwischenverbindung, gemäß einem massensynchronen Parallelschema, wodurch jede der Kacheln in der Gruppe eine kachelinterne Rechenphase gefolgt von einer kachelübergreifenden Austauschphase durchführt, wobei die Austauschphase zurückgehalten wird, bis alle Kacheln in der Gruppe die Rechenphase beendet haben, wobei jede Kachel in der Gruppe beim Beenden der Rechenphase einen lokalen Ausstiegzzustand aufweist; wobei der Befehlssatz einen Synchronisationsbefehl zur Ausführung durch jede Kachel in der Gruppe beim Beenden ihrer Rechenphase umfasst, wobei eine Ausführung des Synchronisationsbefehls die Ausführungseinheit zum Senden einer Synchronisationsanforderung an eine Hardwarelogik in der Zwischenverbindung veranlasst; und

wobei das Verfahren ein Triggern der Logik in der Zwischenverbindung zum Zusammenfassen der lokalen Ausstiegzzustände zu einem globalen Ausstiegzzustand und zum Speichern des globalen Ausstiegzzustands in einem Globalausstiegzzustandsregister auf jeder der Kacheln in der Gruppe umfasst, im Ansprechen auf eine Beendigung der Rechenphase durch alle der Kacheln in der Gruppe, was durch einen Empfang der Synchronisationsanforderung von allen Kacheln in der Gruppe angezeigt ist, sodass der globale Ausstiegzzustand durch einen auf jeder der Kacheln in der Gruppe ablaufenden Codeabschnitt zugänglich ist.

23. Computerprogrammprodukt umfassend einen auf einem Computer lesbaren Speicher verkörpert Code und ausgestaltet zum Ausführen auf dem Verarbeitungssystem nach einem der Ansprüche 1 bis 21, wobei der Code einen Abschnitt zum Ausführen auf jeder Kachel in der Gruppe umfasst, der eine Instanz des Synchronisationsbefehls in jedem Abschnitt enthält.

Es folgen 13 Seiten Zeichnungen

Anhängende Zeichnungen

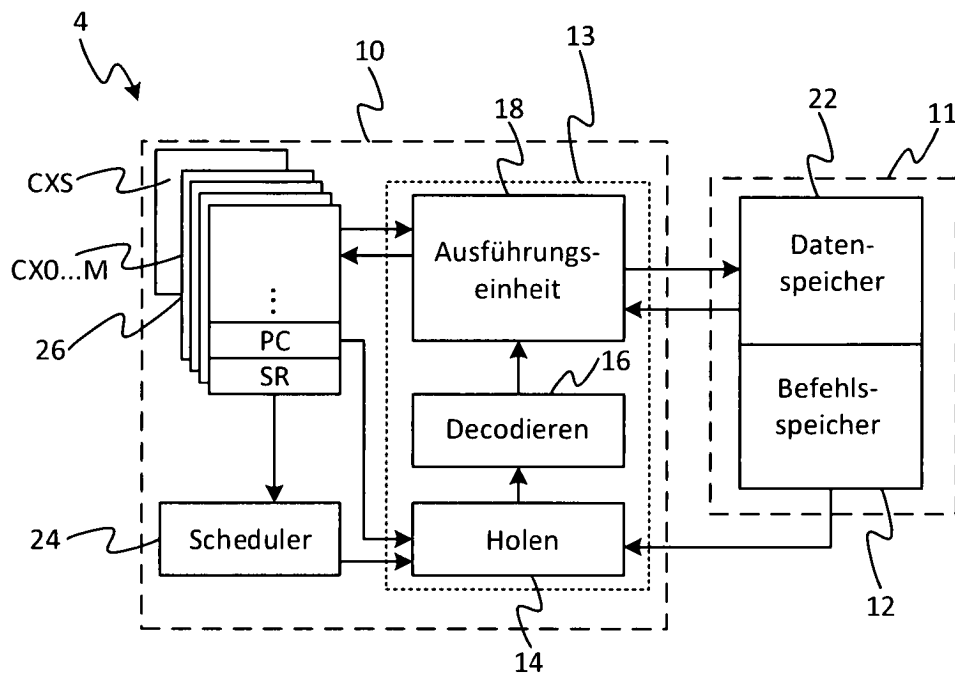


Fig. 1

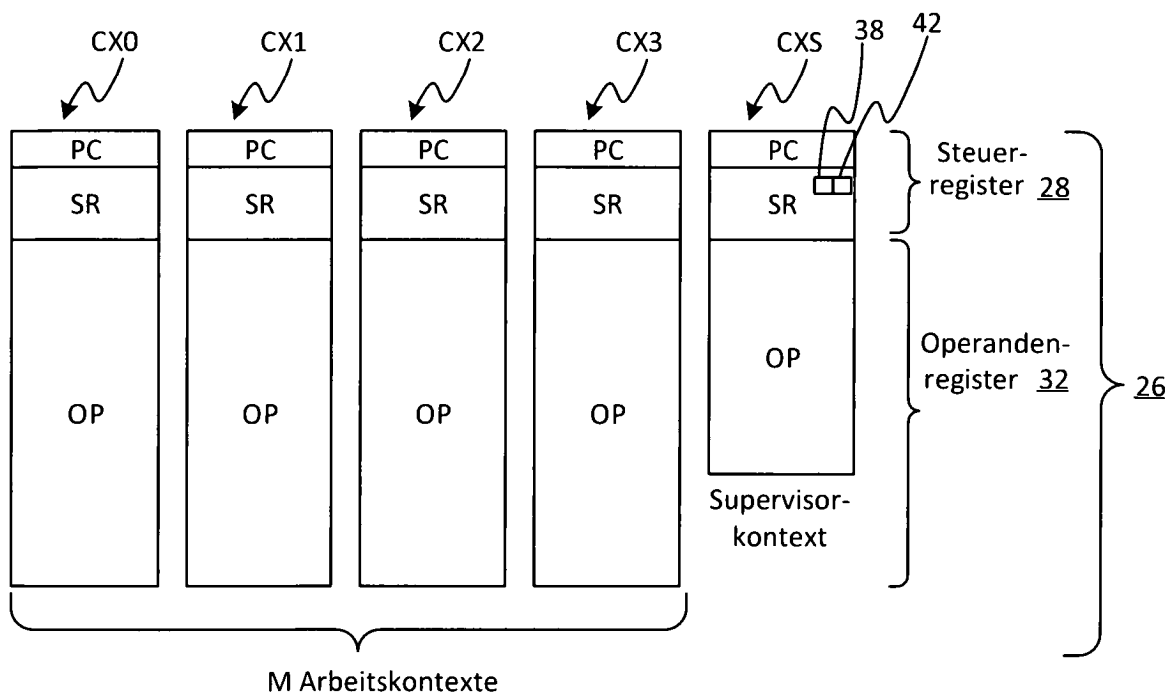


Fig. 2

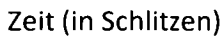


Fig. 3



Fig. 4

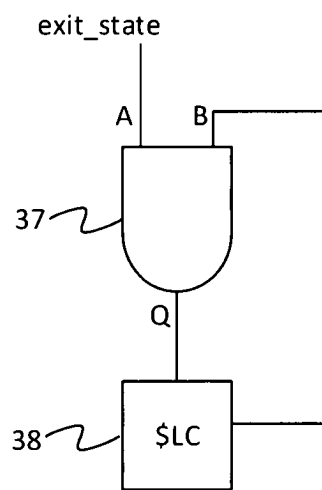


Fig. 5

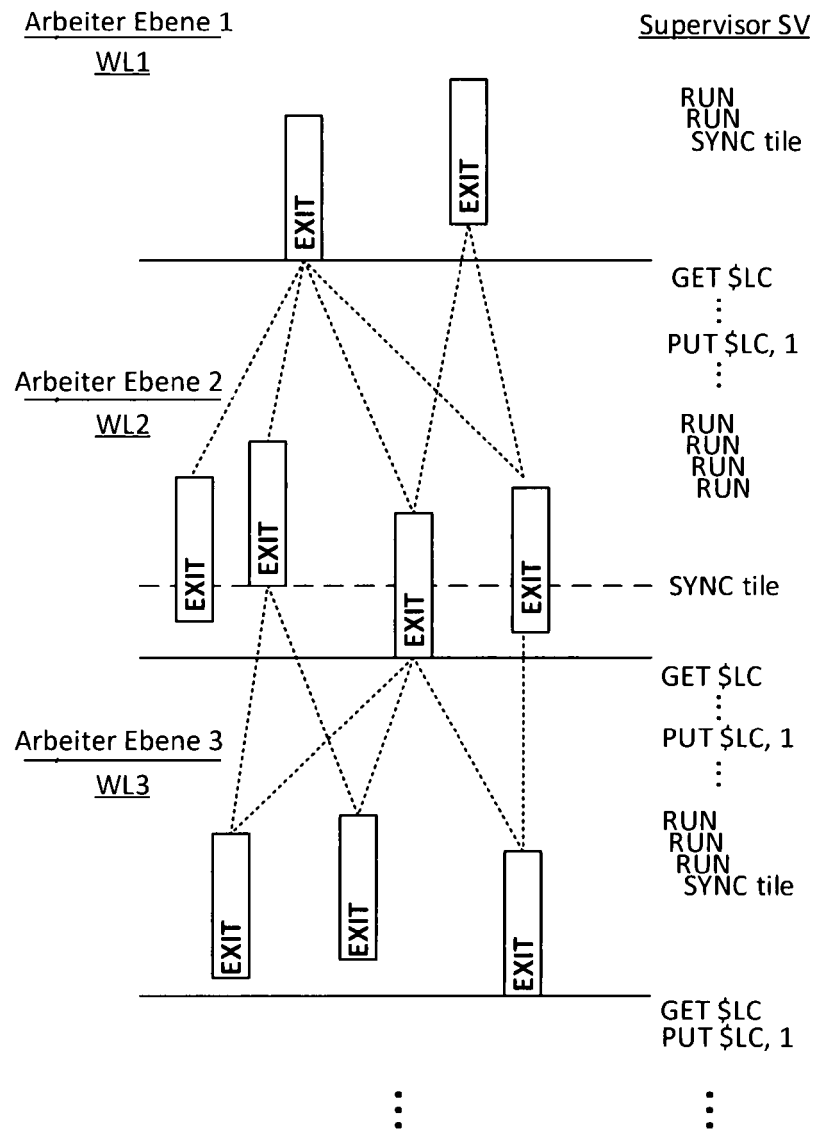


Fig. 6

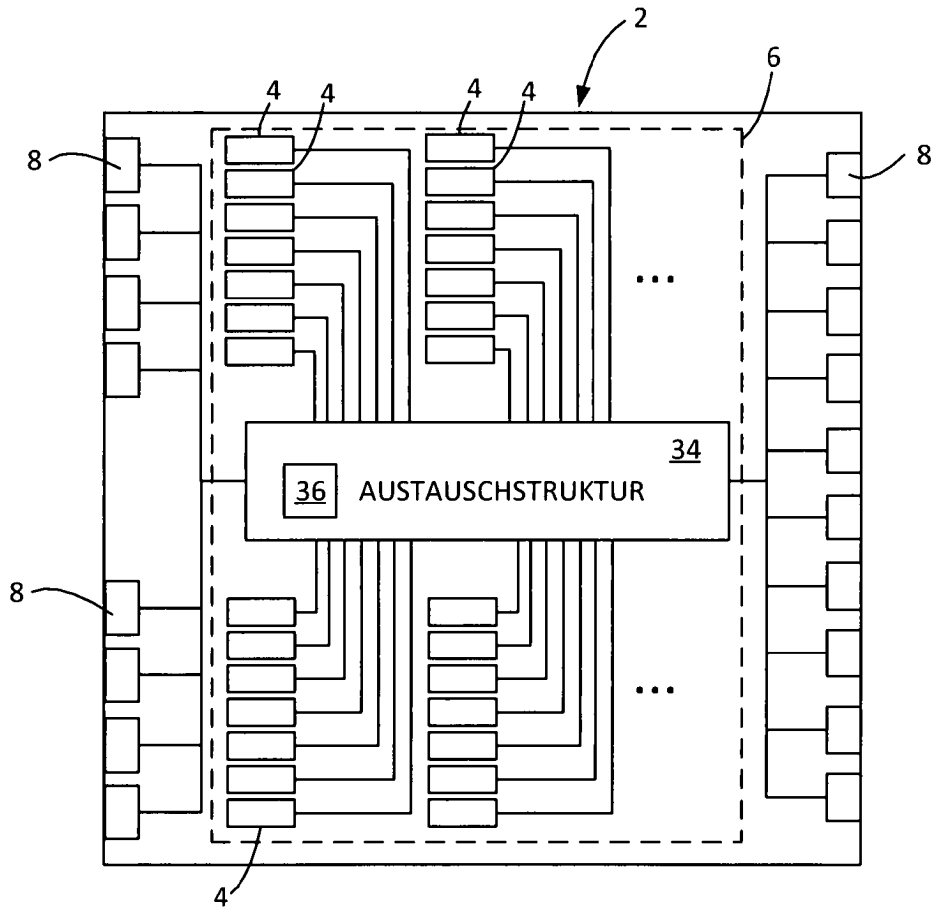


Fig. 7

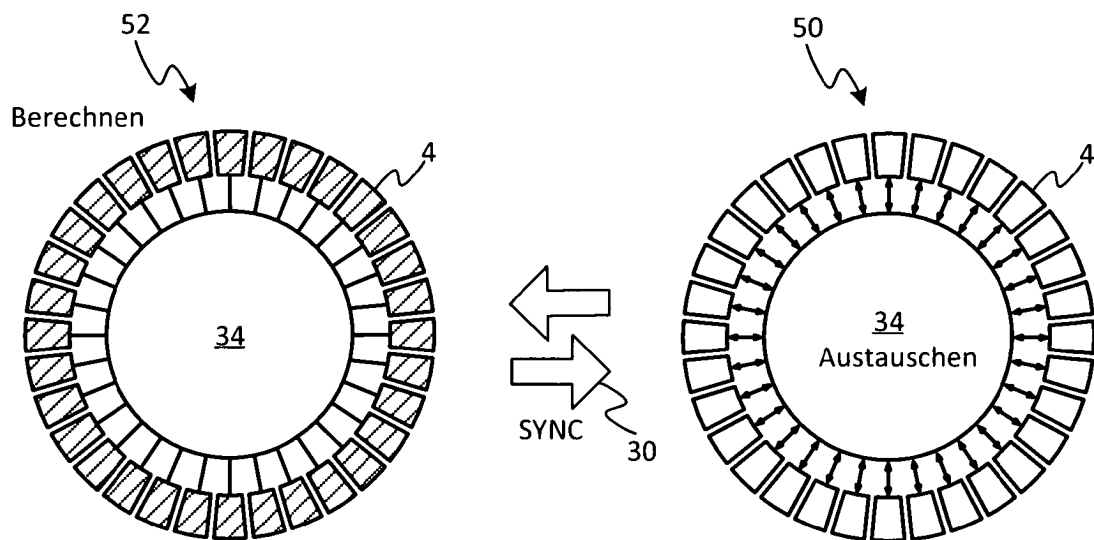


Fig. 8

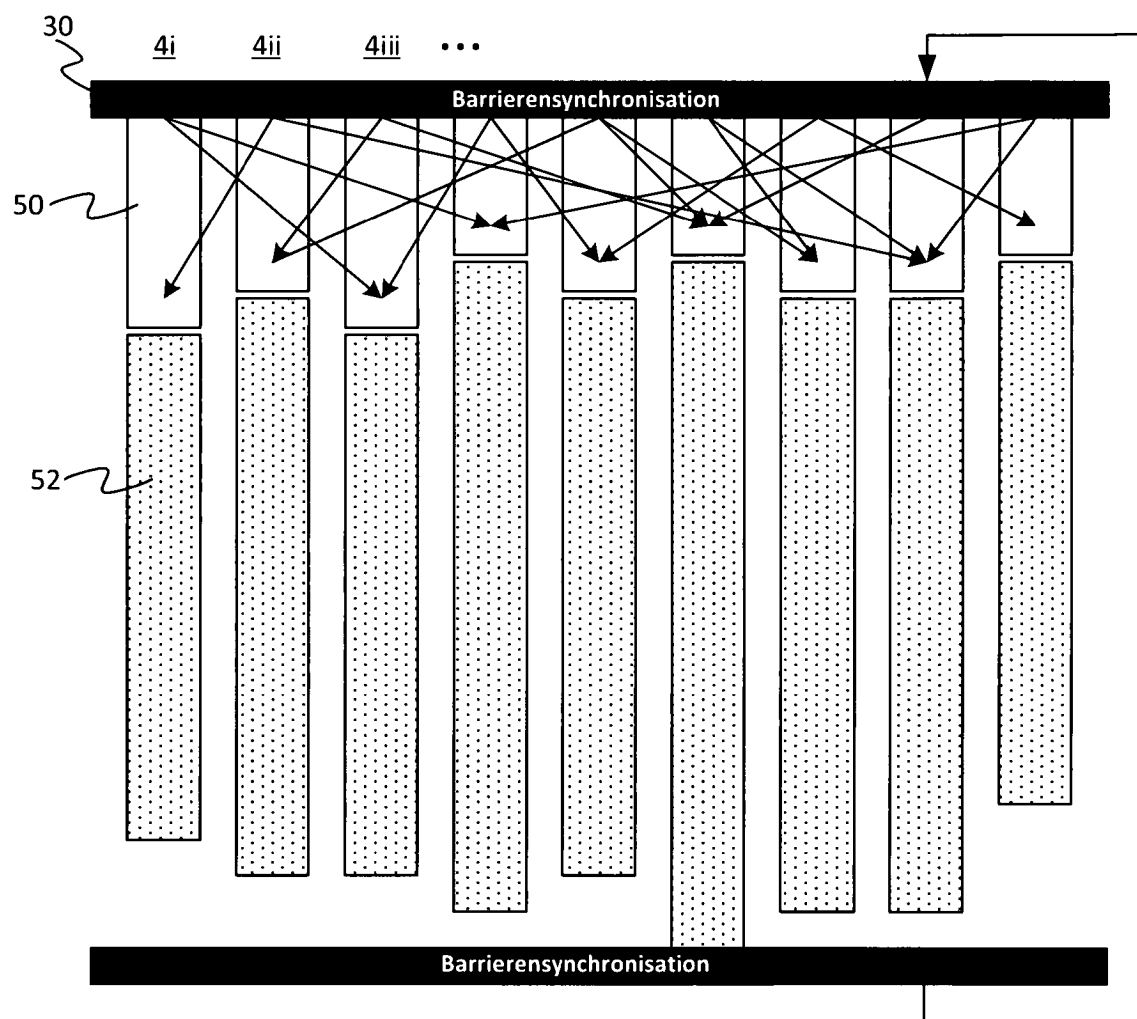


Fig. 9

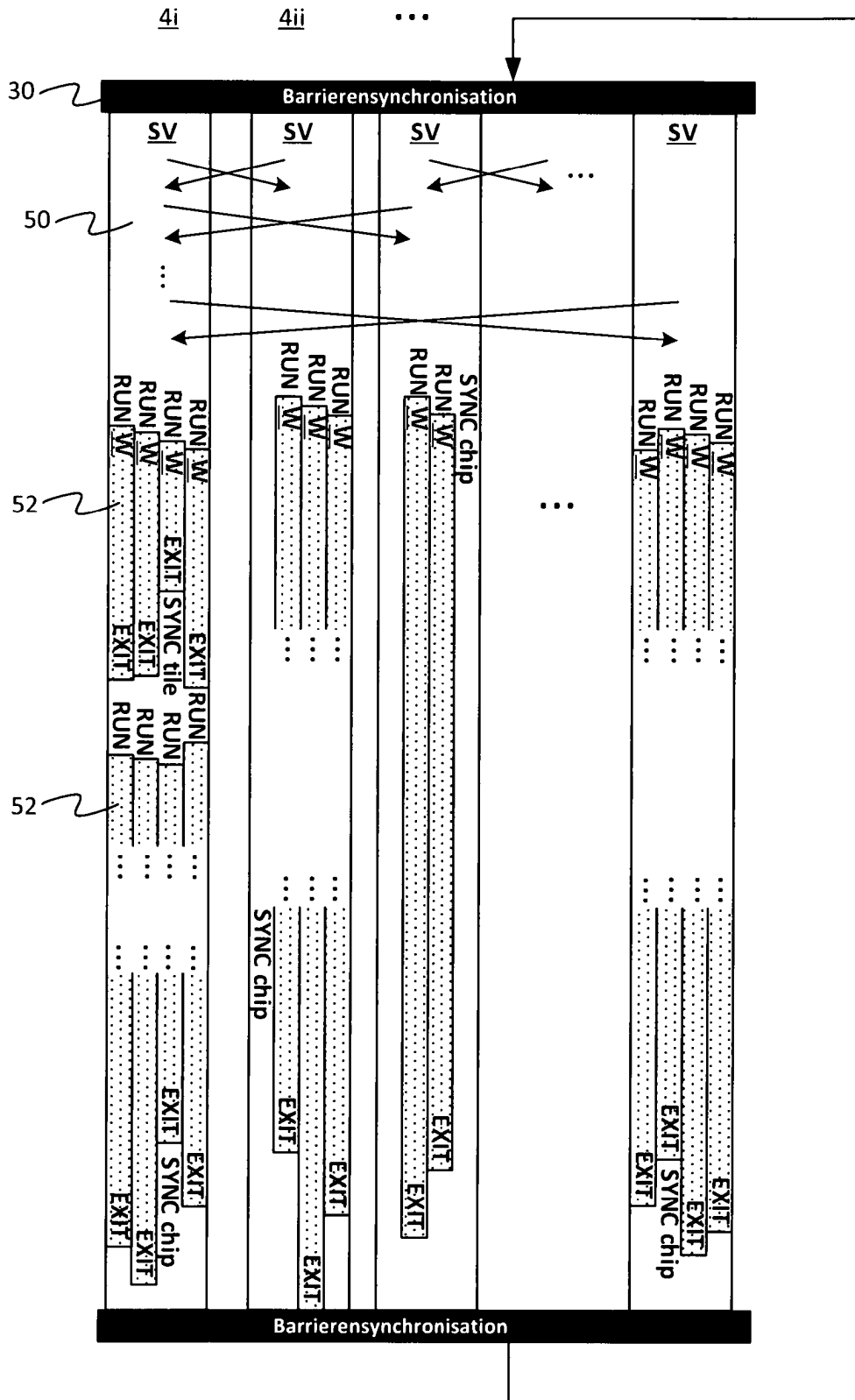


Fig. 10

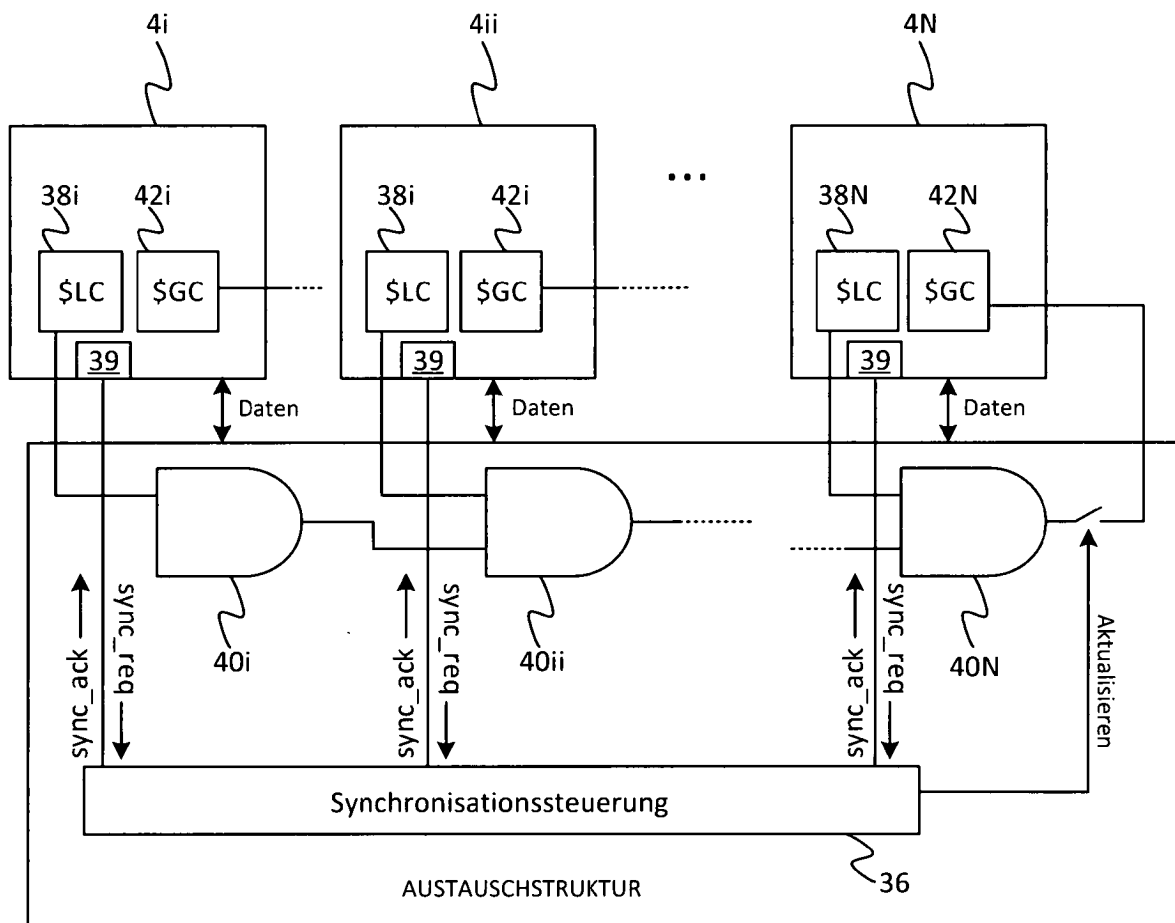


Fig. 11

34

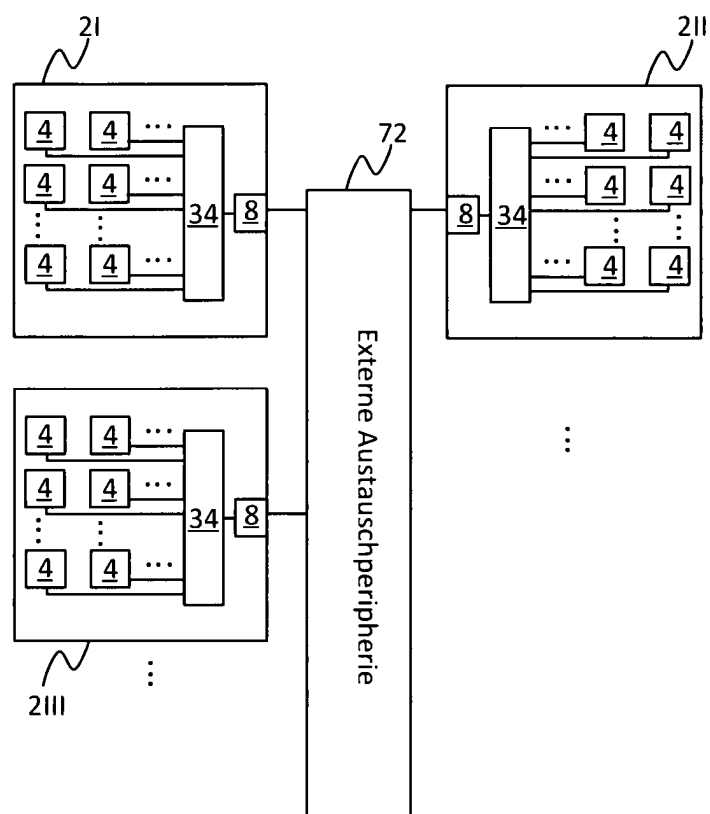


Fig. 12

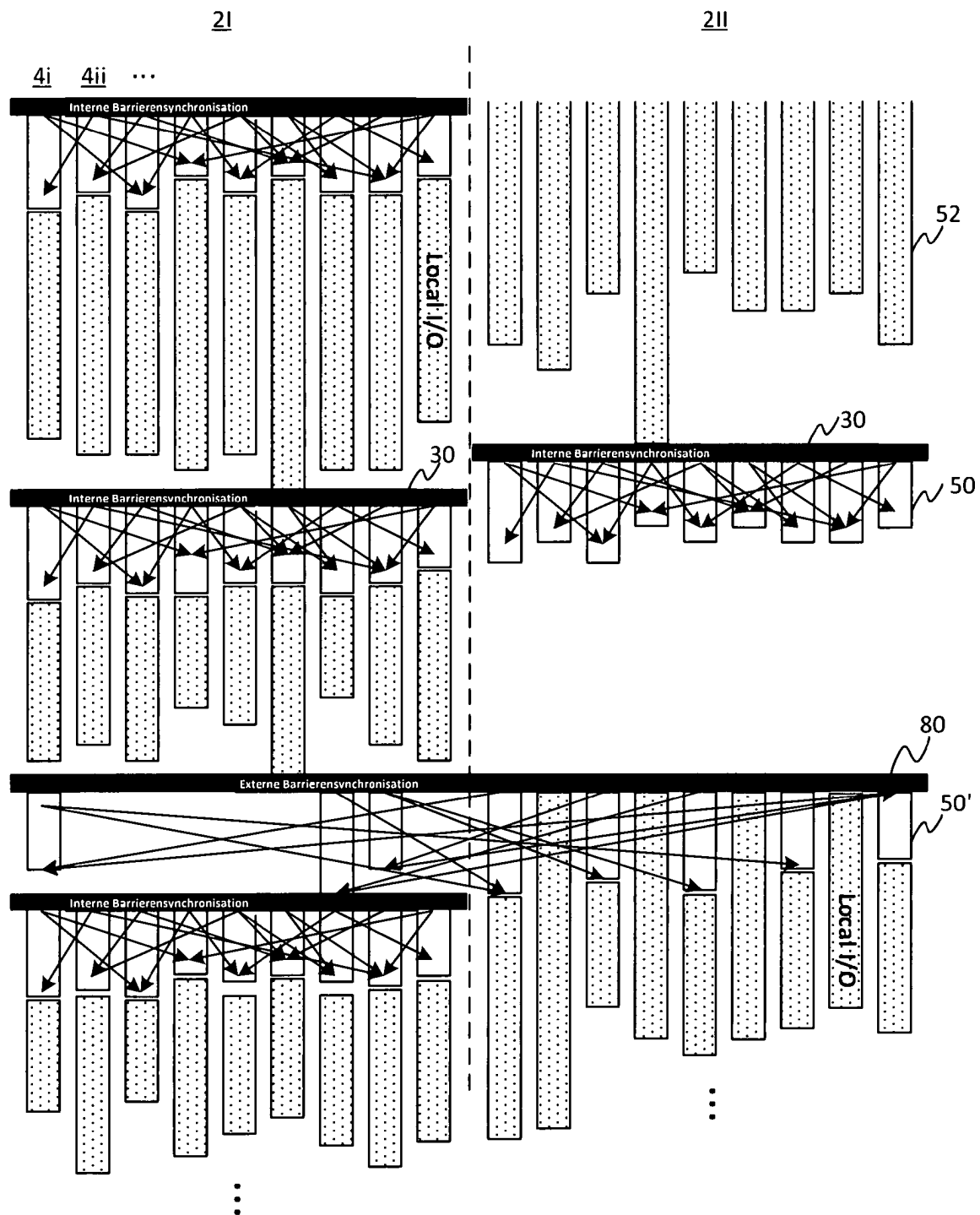


Fig. 13

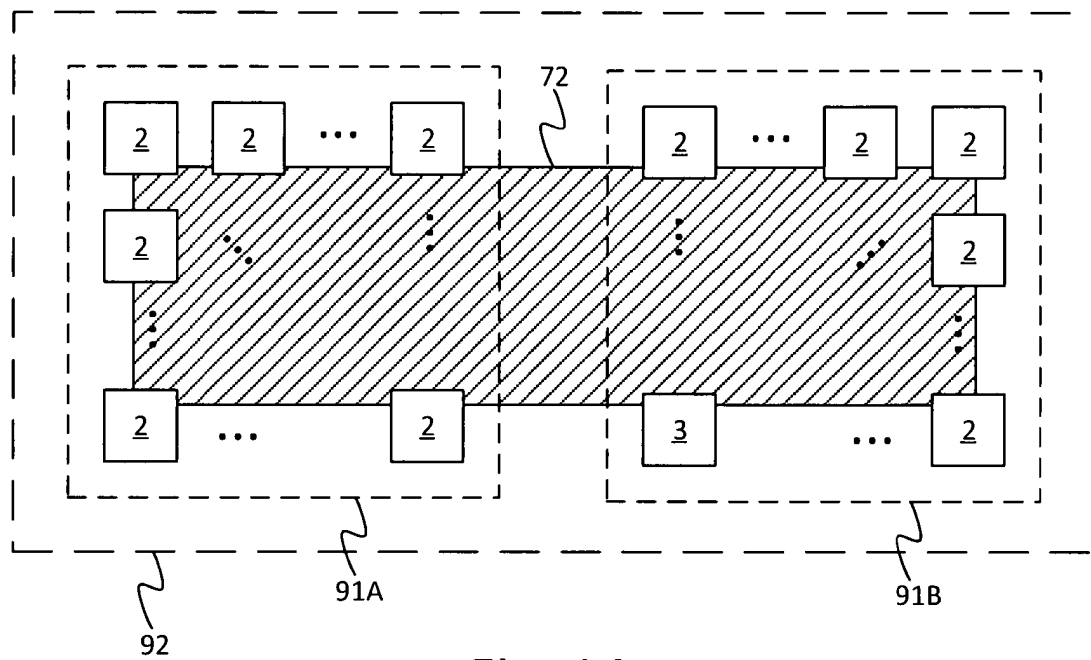


Fig. 14

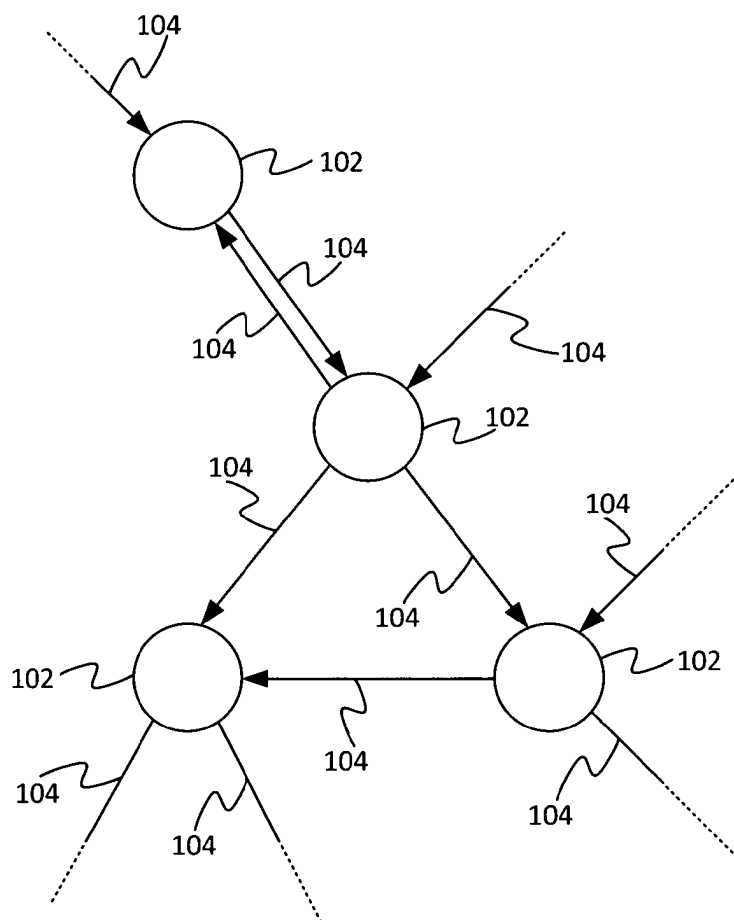


Fig. 15

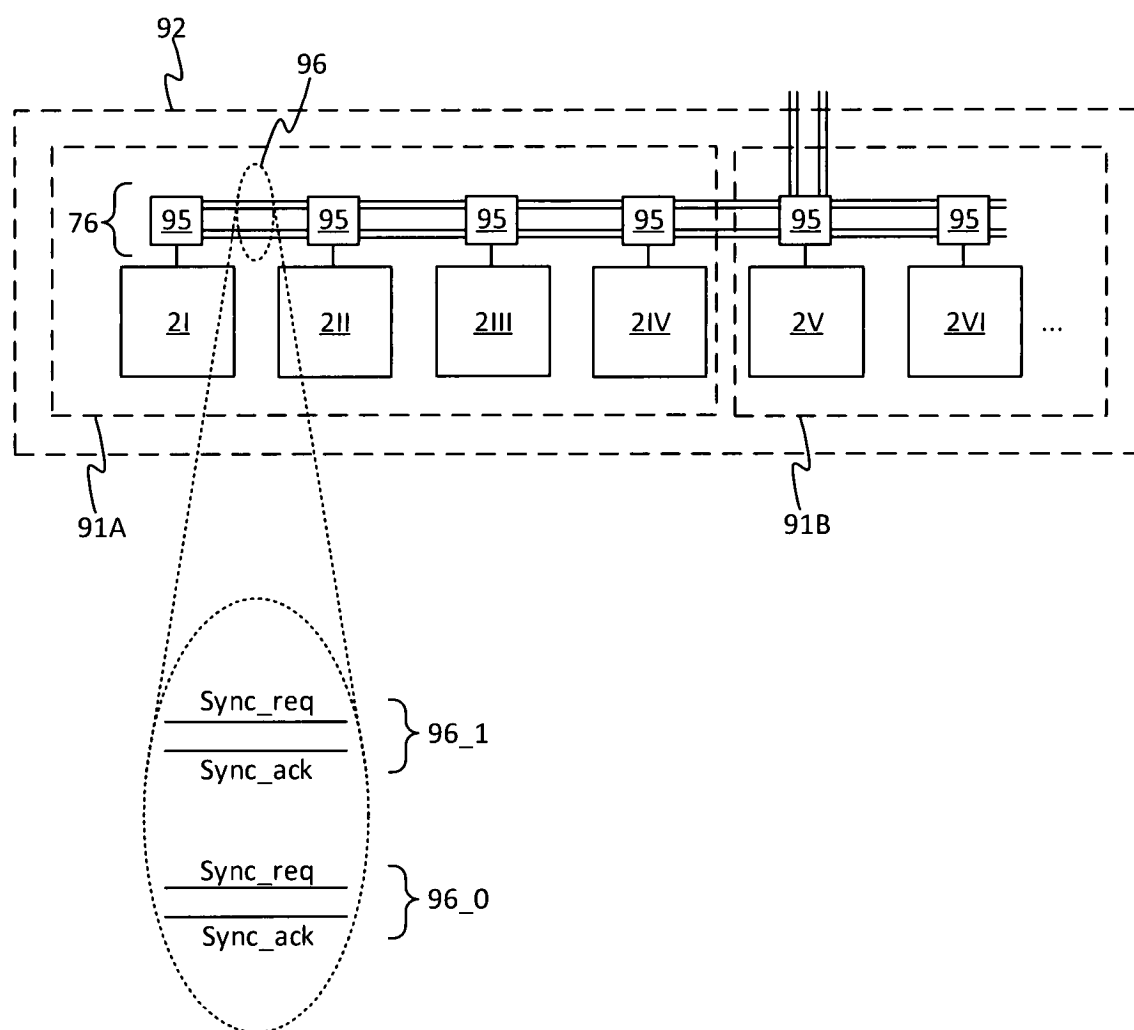


Fig. 16