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

(10) **DE 198 15 865 B4** 2004.11.04

(12) **Patentschrift**

(21) Aktenzeichen: **198 15 865.3**
(22) Anmeldetag: **08.04.1998**
(43) Offenlegungstag: **10.12.1998**
(45) Veröffentlichungstag
der Patenterteilung: **04.11.2004**

(51) Int Cl.[7]: **G06F 9/45**

Innerhalb von 3 Monaten nach Veröffentlichung der Erteilung kann Einspruch erhoben werden.

(30) Unionspriorität:
827619 09.04.1997 US

(71) Patentinhaber:
Ricoh Co., Ltd., Tokio/Tokyo, JP

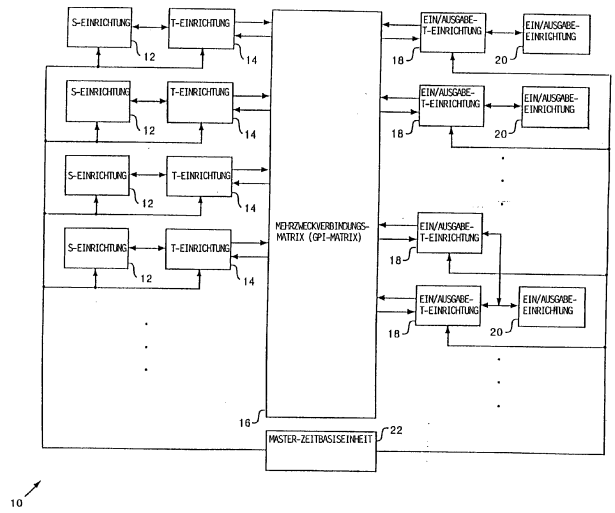
(74) Vertreter:
Schwabe, Sandmair, Marx, 81677 München

(72) Erfinder:
Greenbaum, Jack E., Menlo Park, Calif., US;
Baxter, Michael A., Menlo Park, Calif., US

(56) Für die Beurteilung der Patentfähigkeit in Betracht gezogene Druckschriften:
DE 196 14 991 A1

(54) Bezeichnung: **Kompiliersystem und Verfahren zum rekonfigurierbaren Rechnen**

(57) Hauptanspruch: Kompilierverfahren mittels eines Kompilers zur Erzeugung einer Folge (50) von Programmbefehlen und Rekonfigurations-Anweisungen zur Ausführung in einem dynamisch rekonfigurierbaren Computer (10), der ein Prozessor-Modul (130) aufweist, das ein dynamisch rekonfigurierbares Prozessor-Submodul (12), einen mit dem Prozessor-Submodul (12) verbundenen Programm/Daten-Speicher (133) und einen mit dem Prozessor-Submodul (12) verbundenen Bitstrom-Speicher (132) umfasst, wobei das Prozessor-Submodul durch Laden eines Bitstroms aus dem Bitstrom-Speicher auf Rekonfigurations-Anweisungen hin während der Ausführung der Folge von Programm-Befehlen wahlweise unter einer Anzahl von Befehlssatz-Architekturen (ISA) rekonfiguriert werden kann, mit den folgenden Schritten:
a) als Eingabe wird eine Quelldatei (301) empfangen, die eine Anzahl von Quellcode-Befehlsanweisungen enthält, und zwar einschließlich mindestens eines ersten Untersatzes von Quellcode-Befehlsanweisungen und eines zweiten Untersatzes von Quellcode-Befehlsanweisungen;
b) für den ersten Untersatz von Quellcode-Befehlsanweisungen wird ein erster Befehlsatz, der einer ersten Befehlsatz-Architektur entspricht, mittels einer ersten im Quellcode enthaltenen Rekonfigurations-Übersetzungsanweisung identifiziert, wobei die erste Rekonfigurations-Übersetzungsanweisung den ersten Befehlsatz spezifiziert;
c)...



Beschreibung

[0001] Die vorliegende Erfindung betrifft generell Software für rekonfigurierbare Computer und insbesondere ein Kompiliersystem und ein Verfahren zur Erzeugung ausführbarer Dateien zur Verwendung in einem dynamisch rekonfigurierbaren Computer, der eine veränderbare interne Hardwareorganisation aufweist.

[0002] Ein „dynamisch rekonfigurierbarer Computer" wird im folgenden auch „dynamisch rekonfigurierbare Verarbeitungseinheit" genannt. Weiter wird ein „Prozessor-Modul" auch „Hardwareorganisation" genannt. Ein „Prozessor-Submodul" wird auch „Prozessor-Hardware" genannt. Ein „Bitstrom-Speicher" wird auch „Speicher für Rekonfigurationsbits" und ein „Programm/Daten-Speicher" auch „Speicher für Programme" genannt. Eine „Rekonfigurations-Übersetzungsanweisung" wird auch „Rekonfigurations-Betriebsanweisung" genannt.

Stand der Technik

[0003] Im Stand der Technik sind Versuche unternommen worden, rekonfigurierbare Geräte bzw. Einheiten zu schaffen. Ein erster bekannter Lösungsansatz besteht in herunterladbaren (downloadable) Mikrocode-Geräten, bei denen das Verhalten bzw. der Betriebsablauf von festen, nicht rekonfigurierbaren Hardwarebetriebsmitteln zur Ausführung von Programmen wahlweise verändert werden kann, in dem eine bestimmte Version des Mikrocodes (Programmiersprache für ein Steuerwerk) verwendet wird, der in einen programmierbaren Steuerspeicher geladen wird. Ein Beispiel hierfür findet sich in J.L. Hennessy und D.A. Paterson, Computer Architecture: A Quantitative Approach, Morgan Kaufmann, 1990. In einigen solcher Systeme kann der Mikrocode vom Benutzer nach der Herstellung geschrieben oder verändert werden. Siehe beispielsweise W.T. Wilner „Design of the Burroughs B1700", in AFIPS Fall Joint Computer Conference, AFIPS Press, 1972; W.G. Matheson, „User Microprogrammability in the HP-21MX Minicomputer", in Proceedings of the Seventh Annual Microprogramming Workshop, IEEE Computer Society Press, 1974. Weil die zugrundeliegende Computerhardware in solchen bekannten Systemen nicht selbst rekonfigurierbar ist, zeigen solche Systeme keine optimierte Rechenleistung, wenn man an einen großen Bereich von Problemtypen denkt. Insbesondere sind solche Systeme generell nicht in der Lage, den Datenpfad zu ändern, sind durch die Größe bzw. Speichergröße der Ausführungseinheiten begrenzt und sind nur in der Lage, Wechsel-Befehlssätze (alternate instruction sets) für dieselbe Hardware zu schaffen. Solche Systeme stellen keinen Einzelkompiler zur Verfügung, der in der Lage ist, zwei verschiedene Architekturen zu kompilieren bzw. zu übersetzen.

[0004] Ein zweiter bekannter Lösungsansatz beinhaltet ein System, bei dem die Hardware, die einen Rechenvorgang ausführt, mit Hilfe einer programmierbaren Logik realisiert wird. Es gibt Beispiele hierfür, die feldprogrammierbare Logikschaltungen bzw. feldprogrammierbare Gatearrays (FPGAs) von der Stange verwenden (PAM, SPLASH, VCC) und anwenderprogrammierbare Logik (TERAMAC). Siehe beispielsweise: P. Bertin et al., Programmable Active Memories: A Performance Assessment„ Tech. Rep. 24, Digital Paris Research Laboratory, März 1993; D.A. Buell et al., Splash 2: FPGAs in a Custom Computing Machine, IEEE Computer Society Press, 1996; S. Casselman, "Virtual Computing and The Virtual Computer", in IEEE Symposium on FPGAs for Custom Computing Macines, IEEE Computer Society Press, 1994; R. Amerson et al., "Teramac-Configurable Custom Computing", in IEEE Symposium on FPGAs for Custom Computing Machines, IEEE Computer Society Press, 1995. Im allgemeinen erfordern es diese Technologien, daß eine Anwendung bzw. Applikation hinsichtlich der Hardwarebeschreibung spezifiziert wird, was entweder in Form einer schematischen Beschreibungssprache oder unter Verwendung einer Hardware-Beschreibungssprache, wie beispielsweise VHDL, erfolgt, anstatt daß man Software für einen Computer schreibt, der durch FPGAs festgelegt wird. Beispielsweise wird ein PAM programmiert, indem man ein C++-Programm schreibt, das eine Netzliste erzeugt, die die Konfiguration und Architektur der Gatter bzw. Schaltelemente (Gate) beschreibt. Ein Applikationsentwickler spezifiziert eine Datenstruktur, die eine Hardwarebeschreibung zur Umsetzung der Applikation beschreibt, anstatt daß er eine Spezifizierung eines Applikationsalgorithmus kompiliert. SPLASH wird auf eine der folgenden drei Arten programmiert: 1) Ein schematisches Erfassungspaket (schematic capture package) zum Aufbauen einer Hardwarespezifizierung, basierend auf einem Schaltschema bzw. schematischen Diagramm; 2) einer Hardware-Beschreibungssprache (wie beispielsweise VHDL), die mit einem Synthese-Paket gekoppelt ist, das VHDL in einfache Gate-Anweisungen (primitives) übersetzt; oder 3) einer DBC, d.h. einer C-Untersprache, die in Gate-Beschreibungen kompiliert wird. TERAMAC wird mit Hilfe eines schematischen Erfassungspakets oder einer Hardware-Beschreibungssprache programmiert. Keines dieser Programmierverfahren beschreibt Algorithmenschritte; stattdessen sorgen sie für einen Mechanismus zur Spezifizierung von Hardware-Architekturen.

[0005] Ein dritter bekannter Lösungsansatz beinhaltet rekonfigurierbare Computer, die Softwareprogramme ausführen. Der RISC-4005-Prozessor und der Hokie-Prozessor realisieren Standard-Mikroprozessoren inner-

halb von FPGAs. Der RISC 4005 ist im wesentlichen die Demonstration einer eingebetteten (embedded) zentralen Prozessoreinheit (CPU) innerhalb eines kleinen Abschnittes eines FPGAs, dessen weitere Betriebsmittel (resources) einigen Coprozessor-Funktionen zugeordnet sind. Hokie wird als Lernübung für Informatikstudenten oder Elektrotechnikstudenten verwendet. Eine Befehlsatzarchitektur (instruction set architecture; ISA) wird vor der Kompilierung und Ausführung des Programmes ausgewählt und diese Befehlsatzarchitektur wird dann fortwährend verwendet. Außerdem wird der Bitstrom für den Prozessor separat von der Software abgespeichert, die dieser ausführt. Ad hoc-Verfahren werden verwendet, um sicherzustellen, daß ein korrekter Bitstrom geladen wird. Siehe beispielsweise P. Athanas und R. Hudson, "Using Rapid Prototyping to Teach the Design of Complete Computing Solutions", in IEEE Symposium on FPGAs for Custom Computing Machines, IEEE Computer Society Press, 1996. Diese Systeme sorgen nicht für eine Laufzeit-Rekonfigurierung (während der Ausführung des Programm).

[0006] Bei einem weiteren bekannten, rekonfigurierbaren Computer handelt es sich um den dynamischen Befehlsatz-Computer (Dynamic Instruction Set Computer; DISC), der eine rekonfigurierbare Prozessoreinheit verwendet. Siehe beispielsweise M.J. Wirthlin und B.L. Hutchings, "A Dynamic Instruction Set Computer", in IEEE Symposium on FPGAs for Custom Computing Machines, IEEE Computer Society Press, 1995; D.A. Clark und B.L. Hutchings, "The DISC Programming Environment", in IEEE Symposium on FPGAs for Custom Computing Machines, IEEE Computer Society Press, 1996. Die Ausführung und Konfiguration der FPGA der DISC-Prozessoreinheit wird mit Hilfe eines Mikrocontrollers gesteuert, der ebenfalls in Form eines FPGAs realisiert ist. Der Mikrocontroller wird in einem Dialekt der C-Programmiersprache programmiert. Der Kompiler bzw. das Übersetzungsprogramm für diesen C-Dialekt erkennt, daß gewisse Programmanweisungen durch entsprechende Hardware-Konfigurationen der Verarbeitungseinheit ausgeführt werden sollen und sendet einen Mikrocontroller-Code aus, der veranlaßt, daß der richtige Konfigurations-Bitstrom während der Ausführung des Programms in die Verarbeitungseinheit geladen wird. Der Fachmann auf diesem Gebiet wird erkennen, daß der Mikrocontroller seinerseits einen festen Befehlsatz aufweist und daß der Kompiler diesen festen Befehlsatz kompiliert bzw. übersetzt. Es bestehen mehrere Nachteile hinsichtlich dieser von einem DISC verwendeten Architektur. Weil der Mikrocontroller fest bzw. statisch ist, kann dieser nicht optimiert werden, um verschiedene Arten von Verarbeitungseinheiten zu steuern. Die Konfigurations-Bitströme werden in externer Hardware außerhalb des Speicherplatzes des Mikrocontrollers gespeichert, weshalb das System nicht selbst-enthaltend ist. Außerdem offenbaren die vorstehend im Wege der Bezugnahme in dieser Offenbarung mit beinhalteten Dokumente nicht, wie ein DISC zum Parallelrechnen, zur globalen Signalisierung bzw. systemweiten Datenübermittlung und zum Takten oder zur Handhabung von Interrupts bzw. Unterbrechungsanweisungen verwendet werden könnte. Schließlich werden neue Befehle nur als Einzelgrößen spezifiziert. Der Kompiler sendet nur Befehle für einen Befehlsatz aus, läßt es jedoch zu, daß einzelne Befehle vom Programmierer hinzugefügt werden. Jede Konfigurierung der Verarbeitungseinheit ist ein einzelner Befehl in Form von Hardware, der von dem Programmierer zur Verfügung gestellt wird, wodurch die mögliche Flexibilität eingeschränkt wird.

[0007] Ein vierter bekannter Lösungsansatz besteht darin, Systeme zu mischen, wobei verschiedene Teile des Algorithmus bzw. Rechenvorgangs auf verschiedene Komponenten bzw. Elemente des Systems abgebildet werden. Ein bekanntes System bildet einen Algorithmus, der in einem erweiterten C-Dialekt ausgedrückt ist, auf eine gemischte FPGA/DSP-Architektur ab. Der Benutzer markiert ausdrücklich Abschnitte des Eingabeprogramms zur Zuordnung zum DSP, während der Rest des Codes in Gates zur FPGA-Realisierung hinein kompiliert wird. Solche Systeme erfordern spezialisierte Werkzeuge bzw. Tools, weil sie eine nicht übliche Syntax für ISA-Änderungen verwenden. Außerdem ist der Betrieb solcher Systeme mühsam, was an der Verwendung von Netzlisten zur FPGA-Spezifizierung von Abschnitten des Programms liegt. Solche Programme schaffen keine tatsächliche Hardware-Rekonfigurierung, sondern sorgen lediglich für die Fähigkeit zur Abbildung auf ein anderes Teil der Hardware.

[0008] In gleicher Weise verwenden einige Systeme einen Standard-Mikroprozessor mit einigen konfigurierbaren Logik-Betriebsmitteln. Diese Betriebsmittel (Ressourcen) werden verwendet, um spezielle Befehle bzw. Instruktionen zu realisieren, die die Ausführung von bestimmten Programmen beschleunigen. Siehe beispielsweise R. Razdan und M.D. Smith, "A High-Performance Microarchitecture with Hardware-Programmable Functional Units", in Proceedings of the Twenty-Seventh Annual Microprogramming Workshop, IEEE Computer Society Press, 1994. Solche Systeme werden typischerweise als Zentralprozessoreinheit (CPU) realisiert, und zwar mit einem Abschnitt des Silizium-Chips, der verwendet wird, um einen FPGA zu realisieren. Die CPU besitzt einen festen bzw. fixierten Datenpfad, mit dem die FPGAs verbunden sind. Der Kompiler kombiniert ausgewählte Assembler-Codesequenzen in Einzelbefehlanweisungen zur Ausführung durch ein FPGA. Jedoch arbeiten solche Systeme generell nur auf Ebenen eines bestehenden Assembler-Sprachcodes und erfordern eine angrenzende, feste Befehlsatzarchitektur (nachfolgend als ISA oder auch IS-Architektur bezeich-

net) als Ausgangspunkt. Außerdem halten solche Systeme generell keine Laufzeit-Rekonfigurierung bereit. Schließlich sind solche Systeme nicht weit anwendbar und sorgen üblicher Weise nicht für eine deutliche Geschwindigkeitsverbesserung im Vergleich zu anderen herkömmlichen Systemen.

Aufgabenstellung

[0009] Obwohl die zuvor genannten Systeme jeweils ein gewisses Niveau für die Rekonfigurierbarkeit von Hardware schaffen, beschreibt keines von diesen ein Verfahren oder eine Vorrichtung zur Zusammenfassung von binären Maschinensprache-Befehlen und von Daten gemeinsam mit den Hardware-Konfigurierungen, die notwendig sind, um die Maschinenbebefehle in der in diesem Patent beanspruchten Art und Weise auszuführen. Außerdem offenbaren die bekannten Systeme weder eine Mehrfacharchitektur-ISA-Rekonfiguration auf der Ebene der Granularität, die vergleichbar zu RISC- oder CISC-Befehlen ist, die hierin beansprucht werden, noch Kompilierungsverfahren innerhalb der C-Sprachensyntax zum Ausführen auf dynamisch rekonfigurierten IS-Architekturen, wie diese hierin beansprucht werden.

[0010] Aus DE 19 614 991 A1 ist ein System und Verfahren zum skalierbaren, parallelen, dynamisch rekonfigurierbaren Rechnen bekannt. Insbesondere ist daraus ein dynamisch rekonfigurierbarer Computer bekannt.

[0011] Aufgabe der Erfindung ist es, ein Kompilierverfahren für einen dynamisch rekonfigurierbaren Computer bereitzustellen, wobei insbesondere unterschiedliche Befehlssätze effizient eingesetzt werden sollen.

[0012] Vorstehende Aufgabe wird durch die Gegenstände der Ansprüche 1, 21 und 22 gelöst. Vorteilhafte Weiterbildungen gehen aus den Unteransprüchen hervor.

[0013] Erfindungsgemäß wird ein Verfahren zur Kompilierung von Quellcode geschaffen, der beispielsweise in C oder Pascal geschrieben ist, um ausführbare Dateien bzw. Programme zur Verwendung in einer dynamisch rekonfgurierbaren Prozessoreinheit zu erzeugen, die eine wahlweise veränderbare interne Hardwareorganisation aufweist. Bei einer Ausführungsform können erfindungsgemäß Maschinenbefehle und Daten gemeinsam mit Hardware-Konfigurationen zusammengefaßt werden, welche erforderlich sind, um die Maschinenbefehle auszuführen. In der rekonfigurierbaren Architektur besteht jeder einzelne Prozessor beispielsweise aus: Einer rekonfigurierbaren Prozessor-Hardware, wie beispielsweise einem vollständig FPGA-basierten Prozessor, einem Datenspeicher und einem Programmspeicher, einer Parallel-Verbindungseinheit und einem wiederbeschreibbaren Speicher für FPGA-Konfigurationsbits. Durch dynamisches Laden von FPGA-Konfigurationsbitströmen realisiert die vorliegende Erfindung einen dynamischen ISA-Computer, der eine hohe Leistungsfähigkeit erreicht, in dem ISAs verwendet werden, die für spezielle Phasen der Ausführung der Applikation optimiert sind.

[0014] Bei der Architektur werden Applikationen als Software zur Verfügung gestellt, wird Hardware in der Form von Schaltungen (ein zentrales Service-Modul, Prozessormodule, Eingabe/Ausgabe-Module (I/O)) sowie Bitströme für Befehlsatzarchitekturen (ISAs) bereitgestellt, die auf den FPGAs des Prozessormoduls beheimatet sind. Ein ISA bzw. eine IS-Architektur ist ein primitiver Satz von Befehlen, der dazu verwendet werden kann, einen Computer zu programmieren. Applikationssoftware wird mit Hilfe von FPGAs ausgeführt, die als ISAs auf den Prozessormodulen konfiguriert sind.

[0015] Die vorliegende Erfindung beschreibt ein System, das ausgelegt ist, so daß FPGA-Konfigurationsbitströme statisch während der Kompilierung mit dem Programm verbunden bzw. verknüpft werden können, das diese ausführt, und daß diese zum dynamischen Schalten von ISAs und/oder FPGA-Applikationselementrealisierungen unabhängig und in Echtzeit programmiert werden können.

[0016] Die ISAs führen Programmbefehle aus, die in RAM 133 gespeichert sind. Diese Programmbefehle umfassen wahlweise eine oder mehrere Rekonfigurations-Übersetzungsanweisungen (reconfiguration directives). Bei Auswahl einer Rekonfigurations-Übersetzungsanweisung bzw. Rekonfigurations-Betriebsanweisung wird die Hardware rekonfiguriert, um für eine optimierte Realisierung einer bestimmten Befehlsatzarchitektur zu sorgen. Zusätzlich zu ihrer spezifischen Funktionalität umfaßt jede ISA einen Befehl oder eine Übersetzungsanweisung, der bzw. die veranlaßt, daß eine andere ISA in den rekonfigurierbaren Prozessor geladen wird, so daß die Ausführung der Software anschließend unter Verwendung der neuen ISA fortfährt.

[0017] Weil die Speicherstelle der ISA-Bitströme im Speicher ein Argument für den Rekonfigurationsbefehl darstellt, wird diese Speicherstelle vorzugsweise zum Zeitpunkt der Verbindung (link) oder des Ladens bestimmt, was auch für die Speicherstellen für Funktions-Aufrufziele und -Variablen gilt. Ebenso wie für diese

Funktionen und Variablen hat es sich als wünschenswert herausgestellt, symbolische Namen für die Adressen eines Bitstroms zu verwenden. Die vorliegende Erfindung verwendet ein Objektdatei-Format, das die Vorstellung einer ausführbaren Software auf ISA-Bitströme erweitert. Daraus resultieren einige Vorteile, wie beispielsweise:

– Tools bzw. Werkzeuge können leicht aufgebaut bzw. erstellt werden. Weil die Rekonfigurierung als ein Befehl und Bitströme als Daten behandelt wird, können standardmäßige Software-Verbindungsverfahren eingesetzt werden, um softwaregesteuerte Hardwareänderungen mit den erforderlichen Bitstrom zu binden. Keine neue Softwaretechnologie über die Bitstrom-als-Daten-Abstraktion muß erzeugt werden.

– Flexibilität beim Laden. Indem Rekonfigurationsdaten auf einen Teil des Ausführbaren isoliert werden, wird die Fähigkeit Konfigurationen in geschützte Bereiche des Speichers zu laden, vereinfacht. Mit Gesichtspunkten der Speicherausrichtung wird man auf strukturierte Weise leicht fertig, wie nachfolgend ausführlicher beschrieben werden wird.

– Das Laden wird vereinfacht. Alle Daten, die erforderlich sind, um das Programm auszuführen, werden in einer einzigen Datei aufbewahrt, so daß keine Ladezeit-Identifikation und keine Lokalisierung von Bitströmen ausgeführt zu werden braucht, falls das Ausführbare bzw. das Programm statisch verbunden wird.

– Das Konfigurationsmanagement wird vereinfacht. Nur eine einzige Datei braucht zur gleichen Zeit beibehalten werden, sobald ein Programm gebunden worden ist. Dies vereinfacht die Vorgehensweise zur Verteilung von Applikationen auf einzelne Geräte und entfernte Stellen.

Ausführungsbeispiel

[0018] Nachfolgend wird die Erfindung in beispielhafter Weise und unter Bezugnahme auf die Zeichnungen beschrieben, in denen:

[0019] **Fig.** 1 ein Blockschema der Hardwarekomponenten einer dynamisch rekonfigurierbaren Rechenarchitektur ist;

[0020] **Fig.** 1A ein Blockschema eines erfindungsgemäßen Prozessormoduls ist;

[0021] **Fig.** 1B und 1C Blockschema einer Systemarchitektur zur Realisierung der Erfindung sind, die ein Beispiel für die Rekonfiguration eines FPGAs zeigen;

[0022] **Fig.** 2 ein Beispiel für ein Programmlisting ist, das Rekonfigurations-Übersetzungsanweisungen enthält;

[0023] **Fig.** 3 ein Flußdiagramm für ein Gesamt-Kompilierverfahren ist, das von einem Kompiler bzw. Übersetzungsprogramm zum dynamisch rekonfigurierbaren Rechnen ausgeführt wird;

[0024] **Fig.** 3A und 3B ein Flußdiagramm von bevorzugten Kompiliervorgängen sind, die von einem Kompiler zum dynamisch rekonfigurierbaren Rechnen ausgeführt werden;

[0025] **Fig.** 3C ein Flußdiagramm von weiteren Kompiliervorgängen ist, die von einem Kompiler zum dynamisch rekonfigurierbaren Rechnen ausgeführt werden;

[0026] **Fig.** 4 ein Blockschema eines Kompiliersystems gemäß der vorliegenden Erfindung ist;

[0027] **Fig.** 5 ein Schema eines Objektdateiformats aus dem Stand der Technik ist;

[0028] **Fig.** 6 ein Flußdiagramm für ein Verfahren zum Erhalten eines Programmzustands gemäß der vorliegenden Erfindung ist;

[0029] **Fig.** 7 ein Flußdiagramm für ein Verfahren zur strukturierten Rekonfiguration gemäß der vorliegenden ist; und

[0030] **Fig.** 8A, 8B und 8C Diagramme von Stapelinhalten während einer strukturierten Rekonfiguration gemäß der vorliegenden Erfindung darstellen.

[0031] **Fig.** 9 ein Blockdiagramm einer bevorzugten Ausführungsform eines Systems für ein skalierbares, paralleles, dynamisch rekonfigurierbares Berechnen, gemäß der Erfindung;

[0032] **Fig.** 10 ein Blockdiagramm einer bevorzugten Ausführungsform einer S-Einrichtung gemäß der Erfindung;

[0033] **Fig.** 11A ein beispielhaftes Programmauflisten, das Rekonfigurationsanweisungen enthält;

[0034] **Fig.** 11B ein Ablaufdiagramm von herkömmlichen Kompilieroperationen, die während der Kompilation einer Folge von Programmbefehlen durchgeführt worden sind;

[0035] **Fig.** 11C und 11D ein Ablaufdiagramm von bevorzugten Kompilieroperationen, welche mittels eines Kompilierers für ein dynamisch rekonfigurierbares Berechnen durchgeführt worden sind;

[0036] Die vorliegende Erfindung ist auf ein Kompiliersystem und ein Verfahren zur Erzeugung ausführbarer Dateien zur Verwendung bei einer dynamisch rekonfigurierbaren Prozessoreinheit gerichtet, deren Hardware-Konfiguration nachfolgend insbesondere anhand der **Fig.** 9 bis 11D beschrieben wird.

[0037] In **Fig.** 1 ist ein Blockschema eines skalierbaren, parallelen, dynamisch rekonfigurierbaren Computers **10** zum Ausführen von Objektdateien gezeigt, die gemäß der vorliegenden Erfindung erzeugt wurden. Der Computer **10** umfaßt vorzugsweise mindestens eine S-Einrichtung **12**, eine T-Einrichtung **14**, die jeder S-Einrichtung **12** entspricht, eine Mehrzweck-Verbindungsmatrix (General Purpose Interconnect Matrix; GPI-Matrix) **16**, mindestens eine Eingabe-/Ausgabe-T-Einrichtung **18**, eine oder mehrere Eingabe-/Ausgabeeinrichtungen **20** und eine Master-Zeitbasiseinheit **22**. Bei der bevorzugten Ausführungsform umfaßt der Computer **10** mehrere S-Einrichtungen **12** und somit auch mehrere T-Einrichtungen **14** und außerdem mehrere Eingabe-/Ausgabe-T-Einrichtungen **18** und mehrere Eingabe-/Ausgabeeinrichtungen **20**.

[0038] Jede der S-Einrichtungen **12**, T-Einrichtungen **14** und Eingabe-/Ausgabe-T-Einrichtungen **18** hat einen Mastertakteingang, der mit dem Taktausgang der Masterzeitbasiseinheit **22** verbunden ist. Jede S-Einrichtung **12** hat einen Eingang und einen Ausgang, der mit ihrer entsprechenden T-Einrichtung **14** verbunden ist. Zusätzlich zu dem Eingang und dem Ausgang, die mit der entsprechenden S-Einrichtung **12** verbunden sind, weist jede T-Einrichtung **14** einen Wegsteuerungseingang (routing input) und einen Wegsteuerungsausgang auf, die mit der GPI-Matrix **16** verbunden sind. Dementsprechend hat jede Eingabe-/Ausgabe-T-Einrichtung **18** einen Eingang und einen Ausgang, die mit einer Eingabe-/Ausgabeeinrichtung **20** verbunden sind, sowie einen Wegsteuerungseingang und einen Wegsteuerungsausgang, der mit der GPI-Matrix **16** verbunden ist.

[0039] Bei jeder S-Einrichtung **12** handelt es sich um einen dynamisch rekonfigurierbaren Rechner. Die GPI-Matrix **16** stellt ein paralleles Punkt-zu-Punkt-Verbindungsmittel bzw. ein paralleles Maschen-Verbindungsmittel dar, das die Kommunikation zwischen den T-Einrichtungen **14** erleichtert. Der Satz von T-Einrichtungen **14** und die GPI-Matrix **16** bilden ein paralleles Punkt-zu-Punkt-Verbindungsmittel für einen Datentransfer zwischen Speichern, die der S-Einrichtung **12** zugeordnet sind. In ähnlicher Weise bilden die GPI-Matrix **16**, der Satz von T-Einrichtungen **14** und der Satz von Eingabe-/Ausgabe-T-Einrichtungen **18** ein paralleles Punkt-zu-Punkt-Verbindungsmittel für einen Eingabe-/Ausgabetransfer zwischen S-Einrichtungen **12** und jeder Eingabe-/Ausgabeeinrichtung **20**. Die Master-Zeitbasiseinheit **22** umfaßt einen Oszillator, der jeder S-Einrichtung **12** und jeder T-Einrichtung **14** ein Master-Taktsignal zur Verfügung stellt.

[0040] In einer beispielhaften Ausführungsform ist jede S-Einrichtung **12** durch Verwendung eines Xilinx XC4013 (Xilinx, Inc., San Jose, CA) feldprogrammierbaren Gate-Arrays (FPGA) bzw. Logikanordnung ausgeführt, das mit einem 64 MB Direktzugriffsspeicher (RAM) verbunden ist. Jede T-Einrichtung **14** ist durch Verwendung von annähernd 50 % der rekonfigurierbaren Hardware-Betriebsmittel in einem Xilinx XC4013 FPGA ausgeführt, ebenso jede Eingabe-/Ausgabe-T-Einrichtung **18**. Die GPI-Matrix **16** ist als ein ringförmiges Verbindungsmaschennetz ausgeführt. Die Master-Zeitbasiseinheit **22** ist ein Taktoszillator, der mit einer Taktverteilungsschaltung verbunden ist, um für eine systemweite Frequenzreferenz zu sorgen, wie nachfolgend anhand der **Fig.** 9 bis 25B beschrieben wird. Vorzugsweise übertragen die GPI-Matrix **16**, die T-Einrichtungen **14** und die Eingabe-/Ausgabe-T-Einrichtungen **18** Information entsprechend dem Punkt-zu-Punkt-Protokoll des ANSI/IEEE-Standard 1596-1992, wodurch ein skalierbares kohärentes Interface (SCI) definiert ist.

[0041] In **Fig.** 1A ist ein Blockschema eines Prozessormoduls **130** gezeigt, das in einer Ausführungsform der vorliegenden Erfindung verwendet wird. Der 5-Einrichtungs-FPGA **12** ist mit einem zugeordneten bzw. reservierten Bitstromspeicher **132** und einem Programm-/Datenspeicher **133**, einer oder mehreren T-Einrichtungen **14** und einer Takterzeugungsschaltung verbunden, wie beispielsweise einen Taktgenerator **131**, um ein Prozessormodul **130** zu bilden. Das Modul **130** ist mit anderen, vergleichbaren Modulen über die T-Einrichtungen **14** in einer solchen Art und Weise verbunden, die einen Parallelbetrieb erleichtert. Der Programm-/Datenspei-

cher **133** speichert Programmbefehle und ist in Form eines üblichen RAMs realisiert. Der Bitstromspeicher **132** speichert Bitströme, die die FPGA-Konfigurationen beschreiben. In einer Ausführungsform ist der Programm-/Datenspeicher **133** als dynamisches RAM (DRAM) implementiert und der Bitstromspeicher **132** als statisches RAM (SRAM).

[0042] In den **Fig.** 1B und 1C sind Beispiele für eine FPGA-Rekonfiguration gezeigt, um ISAs in rekonfigurierbarer Architektur zu realisieren. Die Figuren zeigen Blockschema einer Systemarchitektur zur Realisierung der vorliegenden Erfindung, wobei die S-Einrichtungs-FPGA **12** umprogrammiert ist, so daß sie eine arithmetische Logikeinheit (ALU) **143** in **Fig.** 1B umfaßt sowie einen finiten Impulsantwortfilter (FIR) **148** in **Fig.** 1C. Ein Bitstrom-RAM **132** und ein Programm-/Daten-RAM **133** ist vorgesehen. Der Speicherbus **149** hält einen Kommunikationskanal zwischen dem S-Einrichtungs-FPGA **12** und RAM **132** und **133** bereit. Die FPGA-Konfigurationshardware **140** ermöglicht die Rekonfiguration des S-Einrichtungs-FPGAs **12** entsprechend den ISA-Bitströmen vom Bitstrom-RAM **132**. Konfigurationen des S-Einrichtungs-FPGA **12** umfassen beispielsweise Datenregister bzw.

[0043] Datenspeicher **141**, Adreßregister **142**, einen Registermultiplexer **144** und ein Speicherdatenregister **145**. Jede oder alle dieser Komponenten kann modifiziert oder in anderen Konfigurationen entfernt werden, was von dem Bitstrom abhängt. Beispielsweise taucht Alu **143** in der in **Fig.** 1B gezeigten Konfiguration auf, ist aber in der Konfiguration aus **Fig.** 1C durch den FIR-Filter **148** ersetzt.

Rekonfigurations-Übersetzungsanweisungen

[0044] Vorzugsweise speichert der Computer **10** Programmbefehle in RAM wahlweise, und zwar einschließlich von Rekonfigurations-Übersetzungsanweisungen zur Rekonfigurierung von Computer **10**, indem die Konfiguration der S-Einrichtung **12** geändert wird. In **Fig.** 2 ist ein beispielhaftes Programmlisting **50** gezeigt, das einen Satz von Außenschleifenabschnitten **52**, einen ersten Innenschleifenabschnitt **54**, einen zweiten Innenschleifenabschnitt **55**, einen dritten Innenschleifenabschnitt **56**, einen vierten Innenschleifenabschnitt **57** und einen fünften Innenschleifenabschnitt **58** umfaßt. Wie der Fachmann weiß, verweist der Begriff "Innenschleife" auf einen iterativen Abschnitt eines Programms, der dafür verantwortlich ist, einen ganz bestimmten Satz von verwandten Operationen auszuführen; und der Begriff "Außenschleife" verweist auf die Abschnitte eines Programms, die hauptsächlich dafür verantwortlich sind, Mehrzweck-Operationen bzw. universelle Operationen und/oder eine Übertragungssteuerung von einem Innenschleifenabschnitt zum anderen durchzuführen. Im allgemeinen führen die Innenschleifenabschnitte **54** bis **58** eines Programms spezifische Operationen an möglicherweise großen Datensätzen durch. Eine oder mehrere der Rekonfigurations-Übersetzungsanweisungen kann einem vorgegebenen Innenschleifenabschnitt **54**, **55**, **56**, **57** oder **58** zugeordnet sein, so daß sich eine geeignete ISA im Kontext befinden wird, wenn der Innenschleifenabschnitt ausgeführt wird. Im allgemeinen werden für ein beliebiges vorgegebenes Programm die Außenschleifenabschnitte **52** des Programmlistings **50** eine Vielzahl von Mehrzweck-Befehlsarten umfassen, während die Innenschleifenabschnitte **54**, **56** des Programmlistings **50** aus vergleichsweise wenig Befehlsarten bestehen werden, die dazu verwendet werden, einen spezifischen Satz von Operationen auszuführen.

[0045] In einer beispielhaften Programmauflistung **50** erscheint eine erste Rekonfigurations-Übersetzungsanweisung zu Beginn des ersten Innenschleifenabschnitts **54** und erscheint eine zweite Rekonfigurations-Übersetzungsanweisung am Ende des ersten Innenschleifenabschnitts **54**. Dementsprechend erscheint eine dritte Rekonfigurations-Übersetzungsanweisung zu Beginn des zweiten Innenschleifenabschnitts **55**; eine vierte Rekonfigurations-Übersetzungsanweisung erscheint zu Beginn des dritten Innenschleifenabschnitts **56** usw. Jeder Rekonfigurationsbefehl verweist vorzugsweise auf einen Konfigurationsdatensatz, der von einem Bitstrom dargestellt wird. Der Bitstrom spezifiziert eine interne Hardware-Organisation für jede S-Einrichtung **12**, und zwar einschließlich einer dynamisch rekonfigurierbaren Prozessoreinheit (nachfolgend DRPU genannt), einer Adreß-Betriebseinheit (AOU), einer Befehl-Abrufeinheit (IFU) und einer Datenbetriebseinheit (DOU) (nicht gezeigt). Eine solche Hardware-Organisation ist gedacht und optimiert zur Realisierung einer bestimmten Befehlsatzarchitektur (Instruction Set Architecture; ISA). Eine IS-Architektur ist ein einfacher Satz oder Kernsatz von Befehlen bzw. Instruktionen, die dazu verwendet werden können, um einen Rechner zu programmieren. Eine IS-Architektur definiert Befehlsformate, Operationscodes, Datenformate, Adressiermodes, Ausführungs-Steuerflags und programmzugängliche Register bzw. Verzeichnisse. Bei der rekonfigurierbaren Rechnerarchitektur, die zur Ausführung von Objektdateien eingesetzt wird, die gemäß der vorliegenden Erfindung erzeugt werden, kann jede S-Einrichtung sehr rasch und in Echtzeit konfiguriert werden, um unmittelbar eine Folge von IS-Architekturen durch Verwendung eines eindeutigen Konfigurationsdatensatzes für jede gewünschte IS-Architektur zu realisieren, die durch einen Bitstrom spezifiziert wird. Somit wird jede IS-Architektur mit einer speziellen, internen Hardware-Organisation realisiert, wie sie durch einen entsprechenden Konfigurationsdaten-

satz spezifiziert wird. Folglich entsprechen in dem Beispiel aus **Fig.** 2 die ersten fünf Innenschleifenabschnitte **54** bis **58** jeweils einer eindeutigen IS-Architektur 1, 2, 3, 4 bzw. k. Der Fachmann erkennt, daß jede nachfolgende IS-Architektur nicht eindeutig zu sein braucht. Folglich könnte ISA k 1, 2, 3, 4 oder irgendeine andere ISA sein. Der Satz von Außenschleifenabschnitten **52** entspricht ebenfalls einer eindeutigen ISA, nämlich ISA 0. Während der Programmausführung kann die Auswahl nachfolgender Rekonfigurations-Übersetzungsanweisungen von den Daten abhängen. Bei Auswahl einer gegebenen Rekonfigurations-Übersetzungsanweisung werden im Anschluß daran Befehle bzw. Anweisungen nach einer entsprechenden ISA über eine eindeutige 5-Einrichtungs-Hardwarekonfiguration ausgeführt, wie sie durch den Bitstrom spezifiziert wird, auf den durch die Rekonfigurations-Übersetzungsanweisung verwiesen wird.

[0046] Mit der Ausnahme von Rekonfigurations-Übersetzungsanweisungen umfaßt das beispielhafte Programmlisting **50** aus **Fig.** 2 übliche Hochsprachen-Anweisungen, beispielsweise Anweisungen, die entsprechend der C-Programmiersprache geschrieben sind.

[0047] Der Fachmann erkennt, daß der Einbau von einer oder mehreren Rekonfigurations-Übersetzungsanweisungen in einer Folge von Programmanweisungen einen Kompiler bzw. ein Übersetzungsprogramm erfordert, der bzw. das modifiziert wurde, um den Rekonfigurations-Übersetzungsanweisungen Rechnung zu tragen. Folglich umfaßt das erfindungsgemäße Kompiliersystem und das erfindungsgemäße Verfahren Vorgänge einschließlich von Rekonfigurations-Übersetzungsanweisungen durch Zusammenfassung von Verweisungen auf Bitströme, die Hardware-Konfigurationen beschreiben, und durch Übersetzung bzw. Kompilierung von Quellcode entsprechend den Spezifikationen von bestimmten ISAs, die durch die Rekonfigurations-Übersetzungsanweisungen identifiziert werden.

[0048] In einer Ausführungsform der vorliegenden Erfindung unterstützen alle dem Computer **10** zur Verfügung stehende ISAs die folgenden Vorgänge:

– Einen Stapelzeiger (stack pointer; SP) und ein Zeiger Adreßverzeichnis für nächste Befehle (Next Instruction Pointer Address Register; NIPAR; auch bekannt als Programmzähler (PC)), um einen stapel-basierten Speicher von Informationen und Parametern während der Rekonfiguration zu realisieren;

– geeignete Befehle in Assemblersprache zur Flußsteuerung, und zwar einschließlichbeispielsweise von "jsr" bzw. "jump to subroutine" für einen Unterprogramm-Einsprung und "rts" bzw. "return to subroutine" für eine Unterprogramm-Rückkehr; und

– eine geeignete Speicher-Schnittstelleneinheit zum Speichern und Laden von Verzeichniswerten in bzw. aus dem Stapel.

[0049] Die Betriebsweise dieser Komponenten zur Realisierung einer Rekonfiguration wird nachfolgend anhand der **Fig.** 6 bis 8C beschrieben.

Komponenten des Kompiliersystems

[0050] In **Fig.** 4 ist ein Blockschema eines erfindungsgemäßen Kompiliersystems dargestellt. Das Kompiliersystem und das erfindungsgemäße Verfahren läuft auf einer typischen Workstation oder einem PC, der ein übliches Betriebssystem, wie beispielsweise Unix, verwendet. Die Unix-Umgebung ist wegen der großen Verfügbarkeit von Quellcode für Software-Entwicklungstools und der Robustheit der Benutzer-Umgebung vorteilhaft. Wie der Fachmann erkennen wird, könnte das erfindungsgemäße System und das erfindungsgemäße Verfahren direkt auf einem rekonfigurierbaren Computer laufen. In **Fig.** 3 ist ein Flußdiagramm für ein erfindungsgemäßes Gesamtverfahren zur Kompilierung bzw. Übersetzung, zur Assemblierung, zur Verbindung bzw. Verknüpfung und zum Laden gezeigt. Die Kompilierschritte aus **Fig.** 3 werden nachfolgend anhand der **Fig.** 3A bis 3C ausführlicher beschrieben.

[0051] Die Quelldateien **401** werden mit Hilfe eines speziell modifizierten C-Kompilers **402** kompiliert, der nachfolgend beschrieben wird. Der Kompiler **402** liest (**301**) die Quelldateien **401**, die Quellcode-Befehlsanweisungen enthalten, von einem Plattenspeicher oder von einem anderen Eingabe- oder Speichergerät. Der Kompiler **402** identifiziert (**302**) dann eine ISA für einen Untersatz von Quellcodebefehisanweisungen. In einer Ausführungsform werden ISAs von Rekonfigurations-Übersetzungsanweisungen identifiziert, wie nachfolgend ausführlicher beschrieben wird. Der Kompiler **402** erzeugt (**303**) geeignete Rekonfigurations-Übersetzungsanweisungen, um die identifizierte ISA zu spezifizieren, und kompiliert (**304**) den Untersatz von Befehlen zur Ausführung durch die identifizierte ISA, um Anweisungen in Assemblersprache zu erzeugen. Der Kompiler **402** bestimmt dann (**305**), ob ein nachfolgender Untersatz von Befehlsanweisungen (typischerweise eine separate Funktion innerhalb der Quelldatei **401**) mit einer anderen ISA kompiliert werden soll. In einer Ausführungsform wird eine solche Bestimmung wiederum dadurch ausgeführt, daß die Rekonfigurationsübersetzungsanweisun-

gen überprüft werden. Falls eine andere ISA identifiziert wird, kehrt der Kompiler **402** zu Schritt **302** zurück.

[0052] Anderenfalls, wenn das Ende der Quelldatei erreicht wird, werden die Assemblersprachenanweisungen vom Assembler **409** assembliert (**306**), um Objektdateien **403** zu erzeugen. Die Objektdateien **403** werden mit Hilfe des Softwarelinkers bzw. Softwarebinders **404** verbunden (**307**), der modifiziert wurde, um Bitstrom-Speicherstellen und abgeglichene bzw. synchronisierte 64-Bit-Adressen zu behandeln, um ein ausführbares Programm **405** zu erzeugen. Wie nachfolgend beschrieben wird, enthält das ausführbare Programm **405** aufgelöste Verweise (Referenzen) auf ISA-Bitströme **406**, die FPGA-Architekturen festlegen. Nachdem das ausführbare Programm **405** vom Binder **404** erzeugt wurde, wird dieses über die Netzwerkverbindung **408** an das Ladeprogramm **407** gesendet, das auf einem rekonfigurierbaren Computer **10** abläuft, zum Laden (**308**) in den Computer **10**. Für den Fall einer dynamischen Verbindung werden ISA-Bitströme **406** auch über die Netzwerkverbindung **408** an das Ladeprogramm **407** gesendet.

Beliebige und strukturierte Rekonfigurierung

[0053] In einer Ausführungsform läßt der Kompiler **402** eine beliebige Rekonfigurierung (arbitrary reconfiguration) zu, bei der die Rekonfigurations-Übersetzungsanweisungen an einer beliebigen Stelle in dem Quellcode lokalisiert sein können. Bei einer anderen Ausführungsform läßt der Kompiler **402** eine strukturierte Rekonfigurierung (structured reconfiguration) zu, bei der Rekonfigurierungs-Übersetzungsanweisungen nur zugelassen wird, wenn von einer Funktion aufgerufen oder zurückgekehrt wird, so daß jede Funktion mit einer einzelnen ISA bezeichnet wird, die während des gesamten Ablaufs der Funktion im Kontext bzw. Zusammenhang sein soll. Während eine beliebige Rekonfigurierung zusätzliche Flexibilität und einen kleineren Quellcode ermöglicht, sorgt eine strukturierte Rekonfigurierung für eine bessere Vorhersagbarkeit und einen besseren Determinismus beim Laden einer ISA, was zu einer größeren Zuverlässigkeit führt. Weil der Maschinencode generell für verschiedene ISAs verschieden ist, wird der Determinismus bevorzugt, so daß der Kompiler in der Lage ist, einen geeigneten Maschinencode für ein bestimmtes Segment des Quellcodes zu erzeugen. Die beliebige Rekonfigurierung kann zu nicht deterministischen Situationen führen, wenn diese mit gewissen Konditionalkonstrukten im Quellcode kombiniert wird. Diese Situationen werden durch Verwendung einer strukturierten Rekonfigurierung beseitigt.

[0054] Der folgende Auszug aus einem Code stellt ein Beispiel für eine nicht deterministische Rekonfiguration dar, die auftreten kann, wenn eine beliebige Rekonfiguration verwendet wird:

```
#pragma reconfig ISAO
...
x = O
if (a != O) {
#pragma reconfig ISA1
} else {
#pragma reconfig ISA2
}
y = x + 2;
...
```

[0055] Die ISA im Kontext nach der if-Anweisung kann zum Zeitpunkt der Kompilierung nicht bestimmt werden, weil es zur Laufzeit zwei mögliche Pfade für den Steuerfluß gibt, von denen jeder eine Rekonfiguration zu einer anderen ISA bewirkt. Deshalb kann der Kompiler für diese Prozedur keinen gültigen Maschinencode ausgeben, solange ISA1 und ISA2 binär kompatibel sind. Ein solcher Nichtdeterminismus wird beseitigt, wenn eine strukturierte Rekonfiguration verwendet wird, weil nur eine ISA pro Funktion spezifiziert werden kann.

[0056] Bei dem oben genannten Beispiel ist der Wert der Variablen x während des Rekonfigurationsvorgangs geschützt, so daß auf diesen von der neuen IS-Architektur zugegriffen werden kann. In einer Ausführungsform wird der Wert in einem Register bzw. Verzeichnis von ISA0 in herkömmlicher Art und Weise abgespeichert. Die Rekonfigurierung in ISA1 oder ISA2 kann jedoch bewirken, daß dieses Verzeichnis aufhört, zu existieren oder seinen Wert verliert, so daß man sich nicht auf das Verzeichnis verlassen kann, wenn es den Wert von x nach der Rekonfiguration liefert. Der Kompiler **402** überwacht deshalb die lebenden Verzeichniswerte, die nach einer Rekonfiguration verwendet werden, um sicherzustellen, daß ihre Werte zur Verfügung stehen, wenn sie benötigt werden.

[0057] Wenn eine beliebige Rekonfiguration verwendet wird, legt der Kompiler **402** fest, wie eine darauffol-

gende ISA mit einem Zugriff auf eine Variable versehen wird, indem dieser berücksichtigt, wie die augenblickliche ISA die Variable abgespeichert hat, ebenso wie die Einrichtungen, auf die die nachfolgende ISA zugreifen muß. In der strukturierten Rekonfiguration wird der Stapel dazu verwendet, um Werte abzuspeichern, wie dies üblich ist, wenn Werte an eine aufgerufene Funktion übergeben oder von dieser abgerufen werden. Weil sich die Verzeichnisarchitektur während der Rekonfiguration radikal ändern kann, werden lebende Variablen von der scheidenden ISA abgespeichert und dann wieder von der nachfolgenden ISA geladen, wie nachfolgend ausführlicher im Zusammenhang mit **Fig.** 6 erörtert wird.

[0058] Bei einer Ausführungsform realisiert der Kompiler **402** eine bekannte "Linear"-Optimierung (inlining optimization), um eine strukturierte Rekonfiguration zu ermöglichen, um den Aufwand bzw. Systemverwaltungsaufwand der JSR-Anweisung zu vermeiden. Inlining ist ein bekanntes Verfahren zur Optimierung der Kompilierung von Funktionsaufrufen, indem die Vorgänge einer aufgerufenen Funktion "in der Linie" aufgerufen werden, um so den Aufwand zu vermeiden, der mit dem Aufruf der Funktion in üblicher Weise verbunden ist. Somit kann ein Code-Segment, wie beispielsweise:

```
#pragma reconfig ISA1

jsr SUBROUTINE_A
#pragma reconfig ISA0
#pragma reconfig ISA2
jsr SUBROUTINE_B
#pragma reconfig ISA0
```

übersetzt werden durch:

```
#pragma reconfig ISA1
< code von SUBROUTINE_A >
#pragma reconfig ISA0
#pragma reconfig ISA2
jsr SUBROUTINE_B
#pragma reconfig ISA0
```

wodurch die Leistungsfähigkeit verbessert wird, indem die Notwendigkeit einer Sprunganweisung und einer Programmrückkehr umgangen wird und auch die zugeordneten Stapel-Schreibvorgänge, die beim Aufruf einer Funktion und bei der Rückkehr von der Funktion involviert sind.

[0059] Außerdem kann eine zusätzliche Optimierung erfolgen, indem alle Rekonfigurations-Übersetzungsanweisungen bis auf die letzte eliminiert werden, wenn mehr als eine Rekonflgurations-Übersetzungsanweisung in Folge auftritt. Somit kann die dritte Zeile (#pragma reconfig ISA0) von dem oben genannten Code-Segment gelöscht werden.

[0060] Wenn man eine beliebige Rekonfiguration verwendet, kann eine aufgerufene Funktion eine Rekonfiguration bewirken, die wirksam bleibt, auch nachdem der Programmfluß zu der aufgerufenen Funktion zurückkehrt. Beispielsweise beginnt eine aufgerufene Funktion, die die oben genannte Code-Auflistung enthält, in einer ISA, rekonfiguriert zweimal und kehrt dann zu einer aufrufenden Funktion zurück. Von nachfolgenden Anweisungen in der aufrufenden Funktion muß deshalb angenommen werden, daß diese die ISA von der letzten Rekonfiguration verwenden. In einer Ausführungsform führt der Kompiler **402** eine interprozedurale Analyse durch, um zu bestimmen, welche ISAs sich bei jedem Funktionsaufruf und bei jeder Funktionsrückkehr im Kontext befinden. Dort, wo Quelldateien separat in Objektdateien kompiliert werden, bevor diese in ausführbare Programmanweisungen gebunden werden, kann es schwierig oder unmöglich sein, zu bestimmen, welche ISA sich im Kontext befinden wird, nachdem eine aufgerufene Funktion zurückkehrt. In solchen Situationen kann die ISA-Information abgespeichert werden, beispielsweise in Header-Dateien bzw. Kopfinformationsdateien, um zu spezifizieren, welche ISA sich im Kontext bei einer Funktions-Einsprungstelle sowie bei einer Funktions-Austrittsstelle befindet, und zwar für alle externen Funktionen, die von einem Modul aufgerufen werden. Alternativ können Parameter unter Funktionen weitergeleitet werden, um die ISA-Kontexte zu spezifizieren.

[0061] Wenn eine strukturierte Rekonfiguration verwendet wird, ist die ISA-Information im Vereinbarungsteil der Funktion vorgesehen, so daß keine Notwendigkeit besteht, daß der Kompiler **402** ISA-Spezifikationen gegen den Steuerfluß verifiziert, und es gibt keine Möglichkeit, daß eine unerwartete Rekonfiguration während einer aufgerufenen Funktion auftritt.

[0062] Ein weiterer Vorteil einer strukturierten Rekonfiguration besteht darin, daß diese den Rekonfigurationsvorgang vom semantischen Standpunkt her besser wiedergibt. Weil eine Rekonfiguration generell ein Maß an Aufwand mit sich bringt, das zumindest vergleichbar mit dem Aufwand für einen Funktionsaufruf ist, und weil eine Rekonfiguration viele derselben Arten von Operationen beinhaltet, wie beispielsweise die Abspeicherung von Werten auf einem Stapel, ist es wünschenswert, eine ähnliche Syntax sowohl für die Rekonfiguration als auch für Funktionsaufrufe zu schaffen. Die strukturierte Rekonfiguration verbindet die Idee einer Rekonfiguration mit der Idee von Funktionsaufrufen und verwirklicht deshalb dieses semantische Ziel. Weitere Beispiele für eine strukturierte und beliebige Rekonfiguration werden nachfolgend erörtert.

Rekonfigurations-Übersetzungsanweisungen

[0063] Bei der bevorzugten Ausführungsform steht vir Rekonfigurations-Übersetzungsanweisungen #pragma, eine normale Meta-Syntax, die bei der C-Sprache vorgesehen ist, um Information an den Kompiler weiterzuleiten, die aus der Sprachsyntax herausfällt. Die Verwendung der #pragma-Syntax ermöglicht es, daß die Rekonfigurations-Übersetzungsanweisung im Kontext eines C-Programms verwendet werden kann. Ein Beispiel für eine Rekonfigurations-Übersetzungsanweisung, die man in dem Quellcode finden kann, würde wie folgt lauten:
#pragma func-isa func2 isa2

[0064] Bei einer Ausführungsform sind drei #pragma-Übersetzungsanweisungen vorgesehen. Jede Übersetzungsanweisungen wird auf einem anderen Niveau der Granularität oder des Gültigkeitsbereichs betrieben und beeinflußt deshalb einen spezifischen Teil des Codes:

– reconfig: beeinflußt eine Zwischen-Rekonfiguration zu einer neuen ISA (Gültigkeitsbereich ist ein beliebiger Block des Codes);
– func_isa: spezifiziert für eine bestimmte Funktion eine ISA (Gültigkeitsbereich ist die Funktion); und
– default_func_isa: spezifiziert eine Standard-ISA (Gültigkeitsbereich ist die gesamte Datei).

[0065] Diese Rekonfigurations-Übersetzungsanweisungen resultieren in Registertransferniveau-Rekonfigurationsanweisungen (RTL), die den Kompiler mit Information versorgen, um zu bestimmen, welche ISA für jeden Block des Codes benötigt wird, wie nachfolgend ausführlicher beschrieben wird.

[0066] Das nachfolgende Codelisting stellt ein Beispiel für die Verwendung von jeder der oben genannten Übersetzungsanweisungen in einer strukturierten Rekonfigurationsumgebung dar.

```
1     #include "icarus_types.h"
2     #include "icarus_isas.h"
3     #include "fixed.h"
4     #pragma default_func_isa ISA0
5     uns8 color_map[256];

6     #pragma func_isa build_color_map FIXED_POINT_ISA
7     void
8     build_color_map(int16 contrast)
9     {
10          unsigned color;
11          uns8 *color_map_tmp = color_map;

12          for (color = 0; color < 255U; color++) {
13               color_map_tmp[color] =
                      fixed_mul_int8(contrast, color);

14          }
15    }

16    void
17    map_contrast(int x, int y, uns8 *image)
18    {
19          register int i, tmp;

20          tmp = x * y;

21    #pragma reconfig BYTE_MAP_ISA
22          {
23               register int i;
24               register uns8 *map;
25               register uns8 *image_tmp;
26    #pragma isa_pragma map_pointer map
27    #pragma isa_pragma image_pointer image_tmp
28    #pragma isa_pragma loop_counter i

29               image_tmp = image;
30               map = color_map;
31               for (i = tmp; i>0; i--) {
32                    *image_tmp = map[*image_tmp];
33                    image_tmp++;
34               }
35          }
36    }

37    do_contrast(int x, int y, uns8 *image,
          uns8 contrast)
38    {
39          build_color_map(contrast);
40          map_contrast(x, y, image);
41    }
```

[0067] Zeile 4 des Codelistings stellt ein Beispiel für die default_func_ isa-Übersetzungsanweisung dar, die spezifiziert, daß ISA0 für jegliche Funktionen verwendet werden soll, die keine andere ISA spezifizieren. Der Gültigkeitsbereich dieser Übersetzungsanweisung ist die gesamte Datei; deshalb gilt die Übersetzungsanweisung für das gesamte gezeigte Listing.

[0068] Zeile 6 des Codelistings stellt ein Beispiel für die func_isa-Übersetzungsanweisung, die spezifiziert, daß FIXED_POINT_ISA die geeignete ISA für die Funktion darstellt, die build_color_map genannt wird. Der Gültigkeitsbereich dieser Übersetzungsanweisung ist die spezifizierte Funktion.

[0069] Zeile 21 des Codelistings stellt ein Beispiel für die reconfig-Übersetzungsanweisung dar, die spezifi-

ziert, daß BYTE_MAP_ISA die geeignete ISA für den Codeblock darstellt, der unmittelbar der Übersetzungsanweisung folgt. Der Gültigkeitsbereich dieser Übersetzungsanweisung ist der in den Zeilen 22 bis 35 des Codelistings gezeigte Codeblock.

[0070] Das folgende Codelisting stellt ein Beispiel für die Verwendung von jeder der oben genannten Übersetzungsanweisungen in einer beliebigen Rekonfigurationsumgebung dar.

```
1     unsigned char color_map[256];

2     /* int is 16 bits for supercomputer 1A */
3     typedef unsigned int uns16;
4     typedef unsigned char uns8;

5     void
6     build_color_map(uns16 contrast)
7     {
8           int color;

9     #pragma reconfig FixedIsa
10          for (color = 0; color < 256; color++) {
11                color_map[color] =
                        fixed_mul_int8(contrast, color);
12          }
13    }

14    map_contrast(int x, int y, uns8 *image)
15    {
16          register int i, tmp;
17          register uns8 *map;

18          tmp = x * y;
19    #pragma reconfig ByteMapIsa
20    #pragma map_pointer map
21    #pragma map_counter i
22    #pragma target_pointer image
23          i = tmp;
24          map = color_map;

25          while (i--) {
26                *image = map[*image];
27                image++;
28          }
29    }

30    /* Beim Einsprung in do_contrast IsaO annehmen */
31    do_contrast(int x, int y, uns8 *image,
            BinFrac *contrast)
32    {
33          build_color_map(contrast);

34          map_contrast(x, y, image);
35    }
```

[0071] Zeilen 9 und 19 enthalten reconfig-Übersetzungsanweisungen, die solange wirksam bleiben, bis eine andere Rekonfigurations-Übersetzungsanweisung festgestellt wird. Für eine beliebige Rekonfiguration können die Übersetzungsanweisungen an einem beliebigen Punkt in dem Code auftreten und sie sind nicht auf die Funktionsebenen-Granularität begrenzt.

Kompilierungsverfahren

[0072] In den **Fig.** 3A und 3B ist ein Flußdiagramm eines bevorzugten Kompilierungsverfahrens gemäß der

vorliegenden Erfindung gezeigt. **Fig.** 3A zeigt die Schritte, die von der Kompiler-Oberfläche ausgeführt werden, während **Fig.** 3B die Schritte zeigt, die vom Kompiler-Kern ausgeführt werden. Die Oberfläche interpretiert Rekonfigurations-Übersetzungsanweisungen und erzeugt RTL-Anweisungen, die vom Kern in üblicher Weise interpretiert werden können. Wie man weiß, sind RTL-Anweisungen ISA-unabhängige Zwischenniveauanweisungen, die den herkömmlichen Kompilern eingesetzt werden, die beispielsweise in dem GNU C-Kompiler (GCC), der von der Firma Free Software Foundation (Cambridge, MA) hergestellt wird. RTL kann entsprechend der Spezifikation des Stanford University Intermediate Format (SUIF) ausgeführt werden, wie diese in Stanford SUIF Compiler Group, SUIF: A Parallelizing & Optimizing Research Compiler, Tech. Rep. CSL-TR-94-620, Computer Systems Lab, Stanford University, May 1994 offenbart ist. Beispielsweise könnte die Quellcode-Anweisung:

x = y + 3;

in RTL wie folgt dargestellt werden:

r1 <– y

r0 <– r1 + 3

x <– r0

[0073] Das Verfahren aus den **Fig.** 3A und 3B zieht als Eingangsgröße die Quelldatei **401** heran, die eine Folge von Hochsprachen-Quellcodebefehlsanweisungen enthält und die auch mindestens eine Rekonfigurations-Übersetzungsanweisung enthält, die eine ISA zur Ausführung von nachfolgenden Anweisungen spezifiziert. Um dies zu erläutern, sei eine strukturierte Rekonfigurationsumgebung angenommen, bei der eine Rekonfiguration Funktion für Funktion erfolgt. Die Oberfläche des Kompilers **402** wählt (**600**) die nächste Hochsprachenanweisung von der Quelldatei **401** aus und stellt fest (**601**), ob die ausgewählte Hochsprachenanweisung ein Funktionsaufruf ist. Falls dies nicht der Fall ist, sendet (**603**) der Kompiler **402** einen RTL-Code für diese Anweisung.

[0074] Falls der Kompiler **402** in Schritt **601** feststellt, daß die Anweisung ein Funktionsaufruf ist, stellt der Kompiler **402** in Schritt **602** fest, ob die gerade aufgerufene Funktion in einer anderen ISA als der gerade im Kontext befindlichen ISA abläuft. Falls dies nicht der Fall ist, gibt der Kompiler **402** im Schritt **605** einen RTL-Code für den Funktionsaufruf und für das Einlesen des Rückkehrwerts der Funktion in Schritt **613** ab.

[0075] Falls der Kompiler **402** in Schritt **602** feststellt, daß die Funktion in einer anderen ISA arbeitet, gibt der Kompiler **402** einen RTL-Code ab, der erforderlich ist, um die Rekonfiguration zu bewirken, und zwar einschließlich des Abspeicherns aller lebenden Register in Schritt **607** und der Durchführung der Rekonfiguration in Schritt **604**. Bei der bevorzugten Ausführungsform handelt es sich bei der RTL-Rekonfigurationsanweisung um keine Standard-RTL-Anweisung, die eine ISA-Identifikation enthält. Der Kompiler **402** gibt dann in Schritt **606** einen RTL-Code für den Funktionsaufruf ab. Der Kompiler **402** gibt dann in Schritt **609** den RTL-Code für die Rekonfiguration zurück an die erste ISA, um in Schritt **611** lebende Register wieder abzuspeichern und um den Rückgabewert der Funktion in Schritt **613** zu lesen.

[0076] Bei Beendigung der Schritte **603** oder **613** stellt der Kompiler **402** in Schritt **608** fest, ob eine andere Hochsprachenanweisung berücksichtigt werden muß. Falls dies der Fall ist, kehrt der Kompiler **402** zu Schritt **600** zurück; anderenfalls fährt er mit Schritt **610** fort.

[0077] In **Fig.** 3B führt der Kern des Kompilers **402** die Schritte **610** bis **622** aus, um zuvor generierte RTL-Anweisungen in Assemblersprache zu übersetzen.

[0078] Der Kompiler **402** wählt dann in Schritt **612** eine nächste RTL-Anweisung innerhalb der augenblicklich berücksichtigten Gruppe von RTL-Anweisungen aus. Der Kompiler **402** erhält in Schritt **618** eine Regel, die eine Weise spezifiziert, in der die augenblickliche Gruppe von RTL-Anweisungen in einen Satz von Maschinenspracheanweisungen übersetzt werden kann, die für die augenblicklich berücksichtigte Gruppe von RTL-Anweisungen existiert. Der Kompiler **402** erzeugt in Schritt **620** einen Satz von Maschinenspracheanweisungen, die entsprechend der Regel der augenblicklich berücksichtigten Gruppe von RTL-Anweisungen entspricht. Der Kompiler **402** stellt dann in Schritt **622** fest, ob eine andere RTL-Anweisung innerhalb des Kontext einer nächsten Gruppe von RTL-Anweisungen berücksichtigt werden muß. Falls dies der Fall ist, kehrt der Kompiler **402** zu Schritt **612** zurück. Anderenfalls führt der Kompiler **402** in Schritt **610** Registerreservierungs-

schritte (register allocation) aus. Bekanntlich ist eine konsistente Registerarchitektur von einer ISA zur anderen nicht unbedingt erforderlich. Außerdem können gewisse Innenschleifen-ISAs spezielle Register besitzen, für die normale Registerreservierungsvorgänge nicht gelten. Im allgemeinen sind jedoch Außenschleifen-ISAs in der Lage, normale Registerreservierungen zu verwenden.

[0079] Beispielsweise könnte der oben angeführte RTL-Code etwa wie folgt in einen Assembler-Code übersetzt werden, wobei der Assembler-Code von ISA zu ISA verschieden wäre:

```
ld y, r3

ld [r3], r0

add 3, r0

st r0, [x]
```

[0080] Somit erzeugt der Kompiler **402** wahlweise und automatisch in Entsprechung mit Vielfach-ISAs während Kompilierungsvorgängen Assemblersprachenanweisungen. Oder mit anderen Worten: Während des Kompilierungsvorgangs kompiliert der Kompiler **402** einen Einzelsatz von Programmanweisungen von den Quelldateien **401** entsprechend einer variablen ISA. Bei dem Kompiler **402** handelt es sich vorzugsweise um einen üblichen Kompiler, der modifiziert ist, um die bevorzugten Kompilierungsvorgänge durchzuführen, die zuvor anhand der **Fig.** 3A und 3B beschrieben wurden.

[0081] Der Assembler **409** bzw. der Übersetzer für maschinenorientierte Programmiersprache wird betrieben, um Maschinensprachenanweisungen, die vom Kompiler **403** erzeugt wurden, auch dazu zu verwenden, um Objektdateien **403** zu erzeugen. Die Objektdateien **403** werden dann vom Binder bzw. Linker **404** gebunden, der Bitstrom-Speicherstellen und 64-Bit, Bit-ausgerichtete bzw. Bit-abgeglichene Adressen handhabt, um ein ausführbares Programm **405** zu erzeugen. Das Ladeprogramm **407** verkettet gleiche Segmente von einer Anzahl von Objektdateien **403**, einschließlich von Bitstrom-Segmenten, in ein einzelnes Speicherbild zur Übermittlung an einen rekonfigurierbaren Computer **10**. Bei einer Ausführungsform erfolgt eine solche Verkettung während der Laufzeit bzw. in Echtzeit; bei einer alternativen Ausführungsform erfolgt dies off-line. Es ist vorteilhaft, wenn der Binder **404** in der Lage ist, eine Speicherausrichtung bzw. einen Speicherabgleich auf dem ausführbaren Programm **405** auszuführen, um für die Ausrichtungserfordernisse für den FPGA-Bitstrom zu sorgen. Gewisse FPGA-Ladeprogrammhardware erfordert Bitströme von konstanter Größe. Deshalb kann der Binder die Speicherausrichtung vornehmen, indem er Bitströme auffüllt, damit die Anforderungen für eine solche Hardware erfüllt werden.

[0082] Wenn ein statisches Einbinden (static linking) verwendet wird, werden Bitströme **406** und ausführbare Programme **405** vom Binder **404** zur Zeit des Bindens verbunden. Wenn ein dynamisches Binden verwendet wird, werden die ISA-Bitströme **406** und die ausführbaren Programme **405** zum Zeitpunkt des Ladens des Programms verbunden, so daß das ausführbare Programm **405** und die Bitströme **406** über die Netzwerkverbindung **408** zu dem Ladeprogramm **407** gesendet werden, das auf einen rekonfigurierbaren Computer **10** läuft.

[0083] In **Fig.** 3C ist ein Flußdiagramm von weiteren Schritten gezeigt, die zur Erzeugung eines Maschinensprachencodes gemäß einer Ausführungsform der vorliegenden Erfindung ausgeführt werden. Dieses Flußdiagramm gibt im Detail die Zwischendateien an, die erzeugt werden, wenn der RTL-Code in einen maschinenlesbaren Code für einen rekonfigurierbaren Computer übersetzt wird. Der RTL-Code ist mit Bemerkungen versehen **331**, um anzuzeigen, welche ISA sich in Kontext für jede RTL-Anweisung in dem Code befindet. Zu diesem Zeitpunkt werden RTL-Anweisungen modifiziert. Der Kode wird dann in Schritt **332** mit Hilfe von ISA-abhängigen Verfahren und ISA-unabhängigen Verfahren mittels eines Optimierungs-Dienstprogrammes optimiert. Obwohl das Optimierungsdienstprogramm eine ISA-abhängige Optimierung ausführt, verwendet seine Ausgabe maschinenunabhängigen Code. Somit würde die Ausgabe dennoch auf einer beliebigen ISA laufen, obwohl dies nicht notwendiger Weise optimal ist. Schließlich werden maschinenabhängige Befehle im Schritt **333** von dem optimierten Code mittels des Assemblers **409** erzeugt. Dieser Code verwendet abstrakte Register (abstract registers) und andere maschinenabhängige Merkmale. Zusätzliche Schritte zum Aufräumen von Verknüpfungen bzw. Links und weitere unwichtige Schritte können dann ausgeführt werden.

[0084] Bei der bevorzugten Ausführungsform der vorliegenden Erfindung umfassen die ISAs eine reconfig-Anweisung, die bewirkt, daß der FPGA von S-Einrichtung **12** einen Bitstrom lädt, auf den mittels eines Parameters der Anweisung verwiesen wird. Somit besitzt jede ISA zumindest einen Programmverschiebungstyp,

der mit Programmverschiebungsbitstromadressen in Zusammenhang steht, die als Parameter für die reconfig-Anweisung der ISAs verwendet werden. Die Programmverschiebungseinsprungstelle in der Objektdatei teilt dem Binder mit, die augenblickliche Adresse einer Größe in ein Segment eines ausführbaren Programms zum Zeitpunkt der Bindung zu ersetzen. Programmverschiebungstypen werden nachfolgend ausführlicher beschrieben.

[0085] Wie nachfolgend beschrieben wird, werden Bitströme als Datenobjekte definiert, die sich in einem bestimmten Abschnitt befinden, möglicherweise nur zum Lesen, und deshalb sind Standard-Programmverschiebungsverfahren in der Lage, für eine Programmverschiebung von Bitstromadressen zu sorgen, die in analoger Weise mit ISA-reconfig-Anweisungen zu irgendwelchen programmdefinierten, nur lesbaren Daten verwendet werden.

[0086] Der rekonfigurierbare Computer **10** führt die Ergebnisse von dem Ladeprogramm aus, das nachfolgend anhand der **Fig.** 9 bis 25B beschrieben wird. Insbesondere erkennt der rekonfigurierbare Computer **10** reconfig-Anweisungen und lädt geeignete ISA-Bitströme, wie sie in Parametern für solche Anweisungen spezifiziert sind.

Erhaltung des Programmzustands

[0087] Eine FPGA-Rekonfiguration durch Laden einer neuen ISA kann zu einem Verlust von interner Hardwarezustandsinformation führen. Folglich behält das erfindungsgemäße System und das Verfahren den Programmzustand während einer Rekonfiguration bei, um den Verlust der Ausführungsbefehlfolge bei solchen Übergängen der Hardware zu vermeiden.

[0088] Während der Rekonfiguration verwendet der rekonfigurierbare Computer **10** vorzugsweise einen Aufrufstapel, um irgendwelche Daten abzuspeichern, die erforderlich sein könnten, nachdem die neue ISA geladen worden ist. Ein solcher Speichervorgang wird durch Schieben von Werten auf den Aufrufstapel bewerkstelligt und durch Abspeichern des Stapelzeigers in einer vordefinierten Speicherzelle, die nicht durch die Rekonfiguration beeinflußt werden wird. Im Anschluß an die Rekonfiguration verwendet der rekonfigurierbare Computer **10** den Stapelzeiger, um die zuvor abgespeicherten Werte von dem Aufrufstapel auszuspeichern.

[0089] Es sind Stapel in Laufzeitumgebungen für laufende Maschinenprogramme bekannt, die von Hochsprachen kompiliert wurden, die eine Rekursion unterstützen, wie beispielsweise C/C++, Lisp und Pascal. Ein Stapel ist in einem Bereich des Speichers realisiert und der Stapelzeiger (stack Pointer; SP) wird in der ISA dazu verwendet, um die Adresse des Beginns des Stapels zu behalten. Ein Wert, beispielsweise die Programmdaten oder die Adresse, wird in dem Stapel gespeichert (oder auf den Stapel "geschoben"), indem der Stapelzeiger dekrementiert wird und der Wert in die in dem Stapelzeiger enthaltene Adresse geschrieben wird. Der Wert wird wieder von dem Stapel abgerufen (oder vom Stapel "heruntergeschoben"), indem der Wert von der in dem Stapelzeigerregister enthaltenen Adresse gelesen wird; dann wird der Stapelzeiger inkrementiert.

[0090] Bei der vorliegenden Erfindung wird der dynamische Zustand des Programms, wie beispielsweise lokale Variablen und die Speicherstelle der nächsten Anweisung, die die Hardware ausführen soll, die typischerweise in einem Adreßregister für den nächsten Anweisungszeiger (Next Instruction Pointer Address Register; NIPAR) oder in einen Programmzähler (PC) abgespeichert ist, vor der Rekonfiguration der Hardware auf dem Stapel abgespeichert. Der Stapelzeiger wird bei der vorbestimmten Speicheradresse aufbewahrt. Somit werden die Werte des Stapelzeigers und des NIPARs bei der Hardware-Rekonfiguration aufbewahrt, so daß auf diese später Zugriff genommen werden kann, wenn die Ausführung des Programms beginnt.

[0091] In **Fig.** 6 ist ein Flußdiagramm für ein Verfahren zum Konservieren des Programmzustands gemäß der vorliegenden Erfindung gezeigt. Bei Schritt **601** wird eine reconfig-Anweisung empfangen, die anzeigt, daß ein Bitstrom, der eine neue ISA-Konfiguration darstellt, in die Prozessor-Hardware geladen werden soll. Das Argument für die reconfig-Anweisung ist eine physikalische Speicheradresse, die die zu ladende ISA-Konfiguration enthält.

[0092] Der Stapelzeiger wird bei Schritt **652** dekrementiert und der NIPAR wird in Schritt **653** in die von dem Stapelzeiger angezeigte Adresse hineingeladen, wodurch NIPAR auf den Stapel geschoben wird. Der Stapelzeiger wird in Schritt **654** unter einer vorbestimmten Adresse im Speicher abgespeichert, die der neuen ISA-Konfiguration bekannt ist. Die neue ISA-Konfiguration wird dann im Schritt **655** in die Hardware hineingeladen, indem die FPGA **12** dazu veranlaßt wird, den ISA-Bitstrom von einer Speicherstelle in dem Bitstromspeicher **132** zu lesen. Sobald die neue Konfiguration geladen worden ist, lädt diese in Schritt **656** den Stapel-

zeiger von der bekannten, vorbestimmten Adresse und lädt dann NIPAR von dem Stapel, indem dieser von der Speicherstelle abgerufen wird, die in Schritt **657** in dem Stapelzeiger abgespeichert wurde und dann wird der Stapelzeiger in Schritt **658** inkrementiert. Ein Beispiel für Stapelinhalte während des Rekonfigurationsvorgangs aus **Fig.** 6 wird nachfolgend anhand der **Fig.** 8A bis 8C beschrieben.

Realisierung einer strukturierten Rekonfiguration

[0093] Bei einer Ausführungsform der vorliegenden Erfindung wird eine strukturierte Rekonfiguration dadurch bewerkstelligt, daß reconfig-Anweisungen von dem Quellcode in eine Folge von Assemblersprachenanweisungen übersetzt werden. Wie zuvor beschrieben wurde, werden bei der strukturierten Rekonfiguration Rekonfigurations-Übersetzungsanweisungen nur dann zugelassen, wenn eine Funktion aufgerufen wird oder zu einer Funktion zurückgekehrt wird, so daß jede Funktion mit einer einzigen ISA gekennzeichnet ist, die sich während der gesamten Ausführung der Funktion im Kontext befinden soll. In **Fig.** 7 ist ein Flußdiagramm für ein Verfahren zur Realisierung der strukturierten Rekonfiguration gemäß einer Ausführungsform der vorliegenden Erfindung gezeigt.

[0094] Das Verfahren gemäß **Fig.** 7 wird ausgeführt, wenn der Aufruf einer Funktion eine aufgerufene Funktion mit sich bringt, die eine ISA-Rekonfigurationsanweisung besitzt. Die S-Einrichtung **12** rettet in Schritt **707** lebende Registerwerte, so daß diese nicht als Folge der Rekonfiguration verlorengehen. Die S-Einrichtung **12** verwendet die augenblickliche ISA, um in Schritt **701** Parameter für die aufgerufene Funktion auf den Stapel zu schieben, weil diese Parameter in Registern vorliegen können. Die S-Einrichtung **12** rekonfiguriert in Schritt **702** in die neue ISA und ruft in Schritt **703** das Unterprogramm auf, das die Funktion darstellt, wobei eine Anweisung, wie beispielsweise "jsr", der ISA der Zielfunktion verwendet wird. Nachdem die aufgerufene Funktion die Ausführung beendet hat, kehrt eine Rückkehranweisung, wie beispielsweise "ret", in Schritt **704** zu der aufrufenden Funktion zurück. S-Einrichtung **12** 10 rekonfiguriert in Schritt **705** zu der ursprünglichen ISA für die aufrufende Funktion, liest in Schritt **706** einen Rückgabewert der aufgerufenen Funktion und speichert lebende Registerwerte in Schritt **708** wieder ab. Die Verfahren aus **Fig.** 7 zur Weitergabe von Stapelparametern und zur Rückgabe von Werten können in üblicher Weise realisiert werden, wie sie in nicht rekonfigurierbaren Computern verwendet werden, die keine Parameter oder Rückgabewerte in Register weiterleiten.

[0095] Das folgende stellt ein Beispiel für einen Code zur Realisierung einer strukturierten Rekonfiguration dar:

```
#pragma func_isa funcl another_isa
int
func1(int *i)
{
...
}

#pragma func_isa main isa0
main()
{
int foo, bar;
...
bar = func1 (&foo);
}
```

[0096] Zwei Funktionen sind bekannt: main, das ISA0 verwendet, einen Mehrzweck-Anweisungssatz; und func1, das einen anderen Anweisungssatz (another IS) verwendet, der als another_isa bezeichnet wird. Die #pragma-Anweisungen spezifizieren die Anweisungssätze für die zwei Funktionen.

[0097] In einer Ausführungsform der vorliegenden Erfindung, die eine strukturierte Rekonfiguration verwendet, würde der Kompiler **402** die Funktion call bar = func1 (&foo) von dem oben genannten Listing in den folgenden Assemblercode übersetzen. Dabei wurden Kommentare zum Zwecke der Erläuterung hinzugefügt.

```
1  ; schaffe Platz für den Rückgabewert durch Dekrementierung von SP
2  eldi        16, a0
3  esub        a0, sp
4  ; berechne die Adresse von Argument foo
5  emov        sp, a1
6  eadd        a0, a1
7  ; und schiebe diese auf den Stapel
8  estr        a1, sp
9  ; reconfig
10 reconfig    another_isa
```

[0098] Diese Seite bleibt aus technischen Gründen frei.

```
11 ; rufe das Unterprogramm auf
12 jsr         func1
13 ; gehe zurück zu isa0
14 reconfig    isa0
15 ; lösche den geschobenen Parameter
16 eldf        d0
17 ; und lese das Ergebnis in die Registervariable bar
18 ldf         d0
```

[0099] Außerdem könnte Assemblercode zum Abspeichern und wieder Herstellen von lebenden Registerwerten vor der Zeile 1 bzw. nach der Zeile 18 hinzugefügt werden.

[0100] In den **Fig.** 8A bis 8C sind Diagramme der Speicherinhalte bei verschiedenen Punkten während der Ausführung des Assemblercodes gezeigt. Die **Fig.** 8A zeigt den Zustand des Stapels **800** nach der Ausführung der Zeilen 1 bis 8 des oben angeführten Assemblercodes. Diese Zeilen bilden den Stapelrahmen, der von func1 verwendet wird. Zunächst wird Platz geschaffen, um einen Rückgabewert abzuspeichern; dann wird die Adresse der Variablen foo auf den Stapel geschoben. Die Speicherstelle **801** enthält die Variable foo im Stapelrahmen für die Funktion main. In diesem Beispiel wird die Variable bar in einem ISA0-Register abgespeichert und erscheint deshalb nicht auf dem Stapel **800**. Die Speicherstelle **802** wird für einen Rückgabewert reserviert und die Speicherstelle **803** enthält die Adresse der Variablen foo.

[0101] **Fig.** 8B zeigt den Zustand des Stapels **800** auf halben Wege während der Ausführung der reconfig-Anweisung bei Zeile 10. Wie man auch durch Vergleich mit **Fig.** 6 erkennen wird, entspricht dieser Zustand des Stapels **800** dem Ende des Schrittes **654**, unmittelbar bevor die neue Konfiguration geladen werden soll. Die augenblickliche Adresse der nächsten Anweisung (NIPAR) wurde auf den Stapel **800** bei der Speicherstelle **804** geschoben und der Stapelzeiger SP wurde auf eine vorbestimmte Adresse (nicht gezeigt) geschrieben. An dieser Stelle werden die Schritte **655** bis **658** ausgeführt, nämlich die Hardware wird rekonfiguriert, der Stapelzeiger wird geladen und NIPAR wird ausgespeichert, wie zuvor beschrieben wurde.

[0102] **Fig.** 8C zeigt den Zustand des Stapels **800** bei der Einsprungstelle zu func1, wobei die jsr func1-Anweisung bei Zeile 12 verwendet wird. Die Speicherstelle **804** enthält nun die Rückkehradresse. Wenn func1 zurückkehrt, rekonfiguriert der Computer **10** zu ISA0 zurück, wird der Parameter &foo von dem Speicher entfernt und wird der Rückkehrwert in die Variable bar gelesen, die der Kompiler **402** für Register d0 reserviert hatte.

[0103] Die bevorzugte Ausführungsform der vorliegenden Erfindung erweitert Standardparadigmen zur Softwareentwicklung, um Bitströme mit einzuschließen, die Hardwarekonfigurationen festlegen, die einen mit Hilfe von FPGAs realisierten Computer spezifizieren, der binäre Maschinenanweisungen von einer ausführbaren Datei **405** ausführt. Dies wird dadurch bewerkstelligt, daß ein neues Dateiformat verwendet wird, das als ICARUS ELF bezeichnet wird und das eine Erweiterung des Executable and Linking Format (ELF) umfaßt, das häufig auf Unix-Workstations verwendet wird und in UNIX System Laboratories, Inc., System V Application Binary Interface, 3. Auflage, 1993 beschrieben ist und das hiermit im Wege der Bezugnahme in dieser Patentbeschreibung mit aufgenommen sei.

[0104] Wie in dem UNIX System Laboratories, Inc., System V Application Binary Interface, 3. Auflage, 1993 beschrieben ist, handelt es sich bei ELF-Dateien entweder um programmverschiebliche (relocatable) Dateien (Objektdateien **403**) oder um ausführbare Dateien **405**. ELF sorgt für Parallelansichten der Inhalte der Datei, was die differierenden Erfordernisse dieser zwei Formate reflektiert. In **Fig.** 5 ist im Teil **501** ein typisches ELF-Dateiformat in einer Binden-Ansicht und im Teil **502** in einer Ausführen-Ansicht gemäß dem Stand der Technik dargestellt. Der ELF-Kopfteil **503** enthält einen "Plan", der die Organisation der Datei beschreibt. Die Abschnitte **505** beinhalten den Großteil der Information der Objektdatei für die Binden-Betrachtungsweise **501**, einschließlich von Anweisungen, Daten, Symboltabellen, Verschiebungsinformation und dergleichen, wie nachfolgend ausführlicher beschrieben wird. Die Abschnitte **507**, die in der Ausführen-Darstellung **502** verwendet werden, entsprechen den Abschnitten **505**, wobei jeder Abschnitt **507** einem oder mehreren Abschnitten **505** entspricht. Außerdem können die Abschnitte **507** Kopfteile umfassen, die Information enthalten, wie beispielsweise die Information, ob der Abschnitt **507** sich in einem Schreib-Speicher befindet, was auf die Abschnitte **505** anwendbar sein kann oder nicht. Im allgemeinen enthalten die Abschnitte **505** Information, die während des Bindens verwendet wird, während die Abschnitte **507** Information enthalten, die während des Ladens verwendet wird.

[0105] Die Programmkopfteiltabelle **504** (falls vorhanden), teilt dem Computer **10** mit, wie ein Verarbeitungsbild aufzubauen ist. Die Abschnittskopfteiltabelle **506** enthält Information, die die Abschnitte **505** beschreibt. Jeder Abschnitt **505** besitzt einen Eintrag in Tabelle 506; jeder Eintrag gibt Information an, wie beispielsweise den Namen des Abschnitts, die Größe und dergleichen. Die in **Fig.** 5 gezeigten Elemente können in einer beliebigen Reihenfolge vorgesehen sein und einige Elemente können fehlen.

[0106] Weitere Details, die die in **Fig.** 5 gezeigten Elemente betreffen, kann man in UNIX System Laboratories, Inc., System V Application Binary Interface, 3. Auflage, 1993 finden. Die folgende Beschreibung erklärt die Unterschiede zwischen dem Standard-ELF, wie in System V Application Binary Interface beschrieben, und dem ICARUS ELF-Dateiformat, das bei der vorliegenden Erfindung verwendet wird.

[0107] Das ICARUS ELF-Dateiformat verwendet prozessorabhängige Merkmale von ELF, um für eine Verschiebung von Bitstromadressen zu sorgen, die innerhalb des Programmtextes verwendet werden, und um für eine Verschiebung und für ein Binden von Bitströmen in Segmente zu sorgen, die während des Ablaufens des Programms innerhalb eines hierfür vorgesehenen Bitstromspeichers **132** geladen werden können. ICARUS ELF erweitert somit Standard-ELF, um die Abspeicherung von Bitströmen zu erleichtern, die FPGA-Konfigurationen sowie den ausführbaren Code definieren, der auf der FPGA-definierten Hardware läuft.

[0108] ICARUS ELF ergänzt den Standard-ELF, um für neue Datentypen, Abschnitte, Symboltypen und Verschiebungstypen für ISA-Bitströme zu sorgen.

Datentypen

[0109] Bei der bevorzugten Ausführungsform verwendet der rekonfigurierbare Computer Bitadressen, die 64 Bit breit sind. Die Adressen zeigen auf den Bitversatz des niedrigstwertigen Bits des Daten-Gegenstands. ICARUS ELF ist für 64-Bit Byte-Adressen ausgelegt, wobei die Adresse auf das erste Byte (niedrigstwertig für kleine Endian-Prozessoren, höchstwertig für große Endian-Prozessoren) für jedes Datenelement zeigt. Während die Versätze in Kopfteilen bezüglich der Bytes definiert werden, werden zu verschiebende Adressen in 64-Bit Bit-Adressen spezifiziert. Dies läßt die Verwendung eines Binders auf einem Byte-orientierten Computer zu. ICARUS ELF verwendet zwei neue Datentypen, um eine 64-Bitadressierung zu erleichtern:

– ICARUS_ELF_Addr: Größe-8-Bytes, mit Ausrichtung, die für die augenblickliche ISA durch K_{isa} festgelegt wird, das den Zweier-Logarithmus der Bitbreite des Speichers darstellt (beispielsweise 3 für 8-Bit, 4 für 16-Bit).

– ICARUS_ELF_Off: Byte-Versatz in die Datei, Größe 4 Bytes, Ausrichtung 1 Byte.

Abschnitte

[0110] Eine Ausführungsform der vorliegenden Erfindung fügt einen neuen Abschnitt hinzu, der FPGA-Bitstromdaten enthält, mit dem Namen .ICARUS.bitstream. Einer oder mehrere solcher Abschnitte können vorgesehen sein. Bei der bevorzugten Ausführungsform ist jeder solcher Abschnitte vom ELF-Abschnittstyp SHT_PROGBITS und besitzt das ELF-Abschnittsattribut SHF_ALLOC.SHT_PROGBITS bezeichnet einen Abschnitt, der Information enthält, die durch das Programm festgelegt wird, deren Format und Bedeutung ausschließlich über das Programm festgelegt wird. Attribut SHF_ALLOC spezifiziert, daß der Abschnitt während der Vorgangsausführung einen Speicher besetzt, Information, die für das Ladeprogramm nützlich sein kann.

[0111] Weil ELF mehrere Beispiele für einen Abschnitt mit einem bestimmten Namen erlaubt, kann die vorliegende Erfindung einen Abschnitt pro Bitstrom verwenden, oder kann alternativ alle Bitströme in einen Abschnitt mit geeigneter Ausrichtung verbinden.

[0112] Es ist vorteilhaft, für einen neuen Abschnitt für Bitströme zu sorgen, so daß Hardware mit speziellen Speicherbereichen für Bitströme hergestellt werden kann. Der separate Abschnitt erleichtert die Plazierung von Bitströmen in diesen speziellen Speicherbereichen mit Hilfe des Ladeprogramms. Falls solche Speicherbereiche nicht erforderlich sind, kann die vorliegende Erfindung unter Verwendung eines Standard-Datenabschnitts für Nur-Lese-Programmdaten realisiert werden, wie beispielsweise .rodata und .rodata1, wie in System V Application Binary Interface beschrieben ist, anstatt das spezielle Bitstromabschnitte eingeführt werden.

Symbole

[0113] sObjektdateien enthalten Symboltabellen, die Information zur Lokalisierung bzw. Fixierung und zur Verschiebung der Symboladressen und Verweise eines Programms halten. In einer Ausführungsform der vorliegenden Erfindung besitzt jeder Bitstrom, der in dem Abschnitt .ICARUS.bitstream enthalten ist, einen Eintrag in der Symboltabelle der Objektdatei. In der Binden-Ansicht **501** aus **Fig.** 5 ist die Symboltabelle in einem separaten Abschnitt **505** lokalisiert. Das Symbol hat die folgenden Attribute:

– st_narre: Der Name des Symbols ist der Name, der verwendet wird, um es in der Maschinensprachenquelle für die Objektdatei zu referenzieren. st_name enthält einen Index in die Symbolstringtabelle der Objektdatei, die die Zeichendarstellungen der Symbolnamen enthält.

st_value: Sorgt bei Bitstrom-Symbolen für den Versatz des Bitstroms innerhalb des Abschnittes.

st_size: Größe des Bitstroms in Bits.

st_info: Spezifiziert den Typ und die Binde-Attribute. Ein neuer Typ wird verwendet, der als STT_BITSTREAM bezeichnet wird. Dieser neue Typ ist charakteristisch für die vorliegende Erfindung und zeigt an, daß dieses Symbol sich in einem FPGA-Bitstrom befindet. Der Bindevorgang legt die Sichtbarkeit der Bindung und das Verhalten fest und kann STB_LOCAL oder STB_GLOBAL sein. STB_LOCAL zeigt an, daß das Symbol nicht außerhalb der Objektdatei, die die Definition des Symbols enthält, sichtbar ist. STB-GLOBAL zeigt an, daß das Symbol für alle Dateien, die kombiniert werden, sichtbar ist. Für Bitstrom-Symbole kann das Binden entweder STB_LOCAL oder STB-GLOBAL sein. Weil Bitströme für gewöhnlich von mehr als einem Codeabschnitt verwendet werden und deshalb in eine Bibliothek zur Wiederverwendung übersetzt werden können, ist es wahrscheinlicher, daß STB-GLOBAL verwendet wird.

[0114] Verschiebungen (Relocations) Verschiebung ist der Vorgang der Verbindung symbolischer Referenzen mit symbolischen Definitionen. Verschiebbare Dateien enthalten Verschiebungen bzw. Programmverschiebungen, die Daten darstellen, die beschreiben, wo spezielle symbolische Definitionen gefunden werden können, so daß der Binder diese lokalisieren kann. Spezielle Verschiebungsvorgänge variieren von ISA zu ISA, wie dies auch bei Standard-ELF-Dateien der Fall ist. Verschiebungstypen sind innerhalb der Felder r_info von ICARUS_ELF_REL-Strukturen und ICARUS_ELF_RELA-Strukturen enthalten. Beispiele für solche Verschiebungstypen umfassen:

– ICARUS_64_BIT_ADDR: 64-Bitadressen, die zum Zeitpunkt der Übersetzung bestimmt werden. Typischerweise gemeinsam mit der eldi-Anweisung in den Prozessor geladen.

– ICARUS_64_BIT_OFFSET: Relativadreßversatz von augenblicklicher NIPAR-Stelle zu einem Symbol, typischerweise einem Kennzeichen (label). Von den br-Anweisungen (branch; Programmverzweigung) verwendet.

[0115] Aus den genannten Gründen macht erfindungsgemäß das zuvor beschriebene ICARUS ELF-Objektdateiformat neuartigen Gebrauch von der Softwarebindungstechnologie, um Computerprogramme gemein-

sam mit der Hardwarekonfiguration zusammenzufügen, auf der das Programm läuft, wobei ein rekonfigurierbarer Computer **10** verwendet wird, wie er zuvor beschrieben wurde. Das System und das Verfahren gemäß der vorliegenden Erfindung kann eine Kompilierung für Mehrfach-ISAs innerhalb einer einzigen Quelldatei ausführen und ist bei einer Ausführungsform in der Lage, Maschinenanweisungen und Daten gemeinsam mit Hardwarekonfigurationen zusammenzufügen, die erforderlich sind, um die Maschinenin struktionen auszuführen.

[0116] Nachfolgend wird anhand der **Fig.** 9 bis 11D eine bevorzugte Hardware-Umgebung beschrieben, in der die Erfindung bevorzugt angewendet wird.

[0117] In **Fig.** 9 ist ein Blockdiagramm einer bevorzugten Ausführungsform eines Systems **3010** für ein skalierbares, paralleles, dynamisch rekonfigurierbares Berechnen dargestellt, das gemäß der Erfindung ausgeführt wird. Das in. **Fig.** 9 gezeigte System **3010** entspricht im wesentlichen dem in **Fig.** 1 gezeigten System **10**. Das System **3010** weist vorzugsweise zumindest eine S-Einrichtung **3012**, eine T-Einrichtung **3014** entsprechend jeder S-Einrichtung **3012**, eine universelle Verbindungsmatrix (GPIM) **3016**, zumindest eine Ein-/Ausgabe-T-Einrichtung **3018**, ein oder mehrere Ein-/Ausgabevorrichtungen **3020** und eine Master-Zeitbasiseinheit **3022** auf. In der bevorzugten Ausführungsform weist das System **3010** mehrere S-Einrichtungen **3012** und folglich mehrere T-Einrichtungen **3014** plus mehrere Ein-/Ausgabe-T-Einrichtungen **3018** und mehrere Ein-/Ausgabe-Vorrichtungen **3020** auf.

[0118] Jede der S-Einrichtungen **3012**, der T-Einrichtungen **3014** und der Ein-/Ausgabe-T-Einrichtungen **3018** hat einen Master-Zeitsteuereingang, der mit einem Zeitsteuerausgang der Master-Zeitbasiseinheit **3022** verbunden ist. Jede S-Einrichtung **3012** hat einen Eingang und einen Ausgang, der mit der entsprechenden T-Einrichtung **3014** verbunden ist. Zusätzlich zu dem Eingang und dem Ausgang, der mit der entsprechenden S-Einrichtung **3012** verbunden ist, hat jede T-Einrichtung **3014** einen Leiteingang und einen Leitausgang, welche mit der GPI-Matrix **3016** verbunden sind. Dementsprechend hat jede Ein-/Ausgabe-T-Einrichtung **3018** einen Eingang und einen Ausgang, welcher mit einer Ein-/Ausgabe-Vorrichtung **3020** verbunden ist, und einen Leiteingang und einen Leitausgang, der mit der GPI-Matrix **3016** verbunden ist.

[0119] Wie unten im einzelnen noch beschrieben wird, ist jede S-Einrichtung **3012** ein dynamisch rekonfigurierbarer Rechner. Die GPI-Matrix **3016** stellt ein paralleles Punkt-zu-Punkt-Verbindungsmittel dar, welches eine Kommunikation zwischen T-Einrichtungen **3014** erleichtert. Der Satz T-Einrichtungen **3014** und die GPI-Matrix **3016** bilden ein paralleles Punkt-zu-Punkt-Verbindungsmittel für einen Datentransfer zwischen S-Einrichtungen **3012**. In ähnlicher Weise bilden die GPI-Matrix **3016**, der Satz T-Einrichtungen **3014** und der Satz Ein-/Ausgabe-T-Einrichtungen **3018** ein paralleles Punkt-zu-Punkt-Verbindungsmittel für einen Ein-/Ausgabe-Transfer zwischen S-Einrichtungen **3012** und jeder Ein-/Ausgabevorrichtung **3020**. Die Master-Zeitbasiseinheit **3022** weist einen Oszillator auf, der ein Master-Zeitsteuersignal zu jeder S-Einrichtung **3012** und jeder T-Einrichtung **3014** schafft.

[0120] In einer beispielhaften Ausführungsform ist jede S-Einrichtung **3012** durch Verwenden eines Xilinx C4013 (Xilinx, Inc., San Jose, CA) feldprogrammierbaren Gate-Array (FPGA) ausgeführt, das mit einem 64 Megabyte Randomspeicher (RAM) verbunden ist. Jede T-Einrichtung **3014** ist durch Verwenden von annähernd 50% der rekonfigurierbaren Hardware-Ressourcen in einem Xilinx XC4013 FPGA ausgeführt, ebenso jede Ein-/Ausgabe-T-Einrichtung **3018** ist. Die GPI-Matrix **3014** ist als ein ringförmiges Verbindungsmaschennetz ausgeführt. Die Master-Zeitbasiseinheit **3020** ist ein Taktoszillator, der vorgesehen ist, um eine Verteilungsschaltung zu takten, um eine systemweite Frequenzreferenz zu schaffen. Vorzugsweise übertragen die GPI-Matrix **3014** die T-Einrichtung **3012** und die Ein-/Ausgabe-T-Einrichtung **3018** Information entsprechend ANSI/IEEE-Standard 1596 bis 1992, wodurch ein skalierbares kohärentes Interface (SCI) definiert ist.

[0121] In der bevorzugten Ausführungsform weist das System **3010** mehrere S-Einrichtungen **3012** auf, welche parallel arbeiten. Der Aufbau und die Funktionalität jeder der einzelnen S-Einrichtungen **3012** wird im einzelnen anhand von **Fig.** 10 bis 20B beschrieben. In **Fig.** 10 ist ein Blockdiagramm einer bevorzugten Ausführungsform einer S-Einrichtung **3012** dargestellt. Die S-Einrichtung **3012** weist eine erste lokale Zeitbasiseinheit **3030**, eine dynamisch rekonfigurierbare Verarbeitungs-(DRP-)Einheit **3032** zum Ausführen von Programmbefehlen und einen Speicher **3034** auf. Die erste lokale Zeitbasiseinheit **303** hat einen Zeitsteuereingang, welche den Master-Zeitsteuereingang der S-Einrichtung bildet. Die erste lokale Zeitbasiseinheit **3030** hat auch einen Zeitsteuerausgang, der ein erstes lokales Zeitsteuersignal oder ein Taktsignal an einem Zeitsteuereingang der DRP-Einheit **3032** und an einem Zeitsteuereingang des Speichers **3034** über eine erste Zeitsteuerleitung **3040** schafft. Die DRP-Einheit **3032** hat einen Steuersignal-Ausgang, der mit einem Steuersignaleingang des Speichers **3034** über eine Speichersteuerleitung **3042** verbunden ist, einen Adressenausgang, der mit einem

Adresseneingang des Speichers **3034** über eine Adressenleitung **3044** verbunden ist, und einen zweiseitig gerichteten Steuerport, der mit einem zweiseitig gerichteten Steuerport des Speichers **3034** über eine Speicher-Ein-/Ausgabeleitung **3046** verbunden ist. Die DPR-Einheit **3032** hat zusätzlich einen zweiseitig gerichteten Steuerport, der über einen zweiseitig gerichteten Steuerport der entsprechenden T-Einrichtung **3014** über eine externe Steuerleitung **3048** verbunden ist. Wie in **Fig.** 10 dargestellt, überspannt die Speichersteuerleitung **3042** X-Bits; die Adressenleitung **3044** überspannt M-Bits; die Speicherein-/Ausgabeleitung **3046** überspannt (N x k) Bits und die externe Steuerleitung **3048** überspannt Y-Bits.

[0122] In der bevorzugten Ausführungsform empfängt die erste lokale Zeitbasiseinheit **3030** das Master-Zeitsteuersignal bzw. Master-Taktsignal von der Master-Zeitbasiseinheit **3022**. Die erste lokale Zeitbasiseinheit **3030** erzeugt das erste lokale Zeitsteuersignal auf dem Master-Zeitsteuersignal und gibt das erste lokale Zeitsteuersignal an die DRP-Einheit **3032** und den Speicher **3034** ab. In der bevorzugten Ausführungsform kann sich das erste lokale Zeitsteuersignal von einer S-Einrichtung **3012** zur anderen ändern. Folglich arbeiten die DRP-Einheit **3032** und der Speicher **3034** in einer vorgegebenen S-Einrichtung **3012** mit einer unabhängigen Taktrate bezüglich der DRP-Einheit **3032** und dem Speicher **3034** in einer anderen S-Einrichtung **3012**. Vorzugsweise ist das erste lokale Zeitsteuersignal phasensynchronisiert mit dem Master-Zeitsteuersignal. In der bevorzugten Ausführungsform ist die erste lokale Zeitbasiseinheit **3030** durch Verwenden einer phasengekoppelten Frequenzumwandlungsschaltung ausgeführt, die eine phasengekoppelte Detektionsschaltung enthält, die mit Hilfe von rekonfigurierbaren Hardware-Ressourcen augeführt ist. Der Fachmann weiß, daß in einer alternativen Ausführungsform die erste lokale Zeitbasiseinheit **3030** auch als ein Teil eines Taktverteilungsbaums ausgeführt sein könnte.

[0123] Der Speicher **3034** ist vorzugsweise als ein RAM ausgeführt und speichert Programmbefehle, Programmdaten und Konfigurationsdatensätze für die DRP-Einheit **3032**. Der Speicher **3034** einer vorgegebenen S-Einrichtung **3012** ist vorzugsweise für eine andere S-Einrichtung **3012** in dem System **3010** über die GPI-Matrix **3016** zugänglich. Darüber hinaus ist jede S-Einrichtung **3012** vorzugsweise dadurch gekennzeichnet, daß sie einen gleichförmigen Speicheradressenplatz hat. In der bevorzugten Ausführungsform enthalten Programmbefehle, die in dem Speicher **3040** selektiv gespeichert sind, Rekonfigurationsanweisungen, die in Richtung der DRP-Einheit **3032** gerichtet sind.

[0124] In **Fig.** 11A weist die beispielhafte Programmauflistung **3050** einen Satz Außenschleifenteile **3052**, sowie erste bis fünfte Innenschleifenteile **3050** bis **3057** auf. Wie der Fachmann weiß, verweist der Begriff "Innenschleife" auf einen iterativen Teil eines Programms, das dafür verantwortlich ist, einen ganz bestimmten Satz verwandter Operationen durchzuführen, und der Begriff "Außenschleife" verweist auf die Teile eines Programms hin, die hauptsächlich dafür verantwortlich sind, universelle Operationen und/oder eine Übertragungssteuerung von einem Innenschleifenteil zu einem anderen durchzuführen. Im allgemeinen führen Innenschleifenteile **3054** bis **3058** eines Programms spezifische Operationen an möglicherweise großen Datensätzen durch. Bei einer Bildverarbeitungsanwendung kann der erste innere Schleifenteil **3054** Farbformat-Umsetzoperationen an Bilddaten durchführen, und die zweiten bis fünften Innenschleifenteile **3055** bis **3058** können eine lineare Filterung, eine Faltung, Mustersuch- und Kompressionsoperationen durchführen. Wie der Fachmann weiß, kann eine aneinanderhängene Folge von Innenschleifenteilen **3055** bis **3058** als eine Software-Pipeline betrachtet werden. Jeder Außenschleifenteil **3052** würde für eine Daten-Ein-/Ausgabe und/oder für ein Leiten der Datenübertragung und ein Steuern von dem ersten Innenschleifenteil **3054** zu dem zweiten Innenschleifenteil **3055** verantwortlich sein. Zusätzlich erkennt der Fachmann, daß ein vorgegebener Innenschleifenteil **3054** bis **3058** eine oder mehrere Rekonfigurationsanweisungen enthalten kann. Im allgemeinen werden für ein vorgegebenes Programm die Außenscheifenteile **3052** der Programmauflistung **3050** eine Vielfalt von universellen Befehlstypen enthalten, während die Innenschleifenteile **3054**, **3056** der Programmauflistung **3050** aus verhältnismäßig wenigen Befehlstypen bestehen, die verwendet werden, um eine spezifische Menge an Operationen durchzuführen.

[0125] In einer beispielhaften Programmauflistung **3050** erscheint eine erste Konfigurationsanweisung am Anfang des ersten Innenschleifenteils **3054**, und eine zweite Rekonfigurationsanweisung erscheint am Ende des ersten Innenschleifenteils **3054**. Dementsprechend erscheint eine dritte Konfigurationsanweisung zu Beginn des zweiten Innenschleifenteils **3055**; eine vierte Rekonfigurationsanweisung erscheint zu Beginn des dritten Innenschleifenteils **3056**; eine fünfte Rekonfigurationsanweisung erscheint zu Beginn des vierten Innenschleifenteils **3057** und eine sechste und siebte Rekonfigurationsanweisung erscheint am Anfang bzw. am Ende des fünften Innenschleifenteils **3058**. Jede Rekonfigurationsanweisung verweist vorzugsweise auf einen Rekonfigurationsdatensatz, welcher eine interne DRPU-Hardware-Organisation spezifiziert, die auf die Ausführung einer ganz bestimmen Befehlssatz-Architektur (ISA) gewidmet und dafür optimiert worden ist. Eine IS-Architektur ist ein Stamm- oder Kernsatz von Informationen, die verwendet werden können, um einen Rech-

ner zu programmieren. Eine IS-Architektur definiert Befehlsformate, Operationscodes, Datenformate, Adressiermodes, Ausführungs-Steuerflags und programmzugängliche Register. Der Fachmann weiß, daß dies der herkömmlichen Definition einer IS-Architektur entspricht. In der vorliegenden Erfindung kann jede DRP-Einheit **3032** einer S-Einrichtung schnell laufzeit-konfiguriert werden, um direkt Mehrfach-IS-Architekturen durch die Verwendung eines eindeutigen Konfigurationsdatensatzes für jede gewünschte IS-Architektur auszuführen. Das heißt, jede IS-Achitektur wird mit einer eindeutigen internen DRPU-Hardware-Organisation durchgeführt, wie die durch einen entsprechenden Konfigurationsdatensatz spezifiziert ist. Folglich entsprechen in der vorliegenden Erfindung die ersten bis fünften Innenschleifenteile **3054** bis **3058** jeweils einer eindeutigen IS-Architektur, nämlich ISA 1, 2, 3, 4 bzw. k. Der Fachmann erkennt, daß jede nachfolgende IS-Architektur nicht eindeutig zu sein braucht. Folglich könnte ISA k ISA 1, 2, 3, 4 oder irgendeine andere ISA sein. Der Satz Außenschleifenteile **3052** entspricht auch einer eindeutigen ISA, nämlich ISA 0. In der bevorzugten Ausführungsform kann während einer Programmausführung die Auswahl von aufeinanderfolgenden Rekonfigurationsanweisungen datenabhängig sein. Bei Auswahl einer vorgegebenen Rekonfigurationsanweisung werden Programmbefehle nacheinander gemäß einer entsprechenden IS-Architektur über eine eindeutige DRPU-Hardware-Konfiguration ausgeführt, was durch einen entsprechenden Konfigurations-Datensatz spezifiziert ist.

[0126] In der Erfindung kann eine vorgegebene IS-Architektur als eine Innenschleifen-IS-Architektur oder als eine Außenschleifen-IS-Architektur entsprechend der Anzahl und den Typen von Befehlen, welche sie enthält, in Kategorien eingeteilt werden. Eine IS-Architektur, die mehrere Befehle enthält und die zum Durchführen genereller Operationen brauchbar ist, ist eine Außenschleifen-ISA, während eine ISA, die aus relativ wenigen Befehlen besteht und die darauf ausgerichtet ist, spezifische Operationstypen durchzuführen, eine Innenschleifen-ISA ist. Da eine Außenschleifen-ISA darauf gerichtet ist, generelle Operationen durchzuführen, ist eine Außenschleifen-ISA am zweckdienlichsten, wenn eine parallele Programm-Befehlsausführung wünschenswert ist. Die Wirksamkeit einer Ausführung einer Innenschleifen-ISA ist vorzugsweise hinsichtlich Befehlen gekennzeichnet, die pro Taktzyklus durchgeführt werden oder hinsichtlich rechten Ergebnissen gekennzeichnet, die pro Taktzyklus erzeugt worden sind.

[0127] Der Fachmann erkennt, daß die vorhergehende Erörterung einer sequentiellen Programmbefehlsausführung und einer parallelen Programmbefehlsausführung eine Programmbefehlsausführung mit einer einzigen DRP-Einheit **3032** betrifft. Das Vorhandensein von mehreren S-Einrichtung Rekonfigurationsanweisungen **3012** in dem System **3010** erleichtert die parallele Ausführung von mehreren Programmbefehlsfolgen in einer vorgegebenen Zeit, wobei jede Programmbefehlsfolge durch eine vorgegebene DRP-Einheit **3032** durchgeführt wird. Jede DRP-Einheit **3032** ist entsprechend konfiguriert, um eine parallele oder serielle Hardware zu haben, um eine ganz bestimmte Innenschleifen-ISA bzw. eine Außenschleifen-ISA in einer ganz bestimmten Zeit durchzuführen. Die interne Hardware-Konfiguration einer vorgegebenen DRP-Einheit **3032** ändert sich mit der Zeit entsprechend der Auswahl von einer oder mehreren Rekonfigurationsanweisungen, die in eine Folge von durchzuführenden Programmbefehlen eingebettet sind.

[0128] In einer bevorzugten Ausführungsform sind jede IS-Architektur und deren entsprechende interne DRPU-Hardware-Organisation entsprechend ausgelegt, um eine optimale Rechenleistung für eine ganz bestimmte Klasse von Rechenproblemen bezüglich einer Menge verfügbarer rekonfigurierbarer Hardware-Ressourcen zu schaffen. Wie vorher bereits erwähnt und wie nunmehr nachstehend im einzelnen näher beschrieben wird, ist eine interne DRPU-Hardware-Organisation, die einer Außenschleifen-ISA entspricht, vorzugsweise für eine sequentielle Programmbefehlsausführung optimiert, und eine interne DRPU-Hardware-Organisation, die einer Innenschleifen-ISA entspricht, ist vorzugsweise für eine parallele Programmbefehlsausführung optimiert.

[0129] Mit Ausnahme jeder Rekonfigurationsanweisung weist die beispielhafte Programmauflistung **3050** der **Fig.** 11A vorzugsweise herkömmliche Hochsprachenangaben auf, beispielsweise Angaben die entsprechend der C-Programmiersprache geschrieben sind. Der Fachmann erkennt, daß das Einbeziehen von einer oder mehreren Rekonfigurationsanweisungen in eine Folge von Programmbefehlen einen Compiler erfordert, der modifiziert ist, um für die Rekonfigurationsanweisungen verantwortlich zu sein.

[0130] In **Fig.** 11B ist ein Flußdiagramm von herkömmlichen Compilieroperationen dargestellt, die während des Compilierens bzw. Übersetzens einer Folge von Programmbefehlen durchgeführt worden sind. Hierbei entsprechen die herkömmlichen Compilieroperationen im allgemeinen denjenigen, die von dem GNU C Compiler (GCC) durchgeführt worden sind, der von der Free Software Foundation (Cambridge, MA) hergestellt worden ist. Der Fachmann weiß, daß die herkömmlichen Compilieroperationen, die unten beschrieben werden, ohne weiteres für andere Compiler verallgemeinert werden können. Die herkömmlichen Compilieroperationen beginnen beim Schritt **3500** mit dem Compiler-Frontende, das eine nächste Hochsprachen-Anweisung für eine

Folge von Programmbefehlen auswählt. Als nächstes erzeugt das Compiler-Frontende beim Schritt **3502** einen Zwischencode, der der ausgewählten Hochsprachen-Anweisung entspricht, welche im Falle von GCC Register-Transferpegel-(RTL-)Angaben entspricht. Im Anschluß an den Schritt **3502** bestimmt das vordere Compilerende, ob eine andere Hochsprachen-Anweisung eine Beachtung beim Schritt **3504** erfordert. Wenn dem so ist, kehrt das bevorzugte Verfahren auf den Schritt **3500** zurück.

[0131] Wenn beim Schritt **3504** das vordere Compilerende bestimmt, daß keine andere Hochsprachenanweisung Beachtung erfordert, führt das hintere Compilerende als nächstes herkömmliche Registerzuordnungsoperationen beim Schritt **3605** durch. Nach dem Schritt **3506** wählt das hintere Compilerende eine nächste RTL-Angabe hinsichtlich einer aktuellen RTL-Anweisungsgruppe beim Schritt **3508** aus. Das hintere Compilerende bestimmt dann, ob eine Vorschrift, die eine Art und Weise spezifiziert, in welcher die aktuelle RTL-Anweisungsgruppe in einen Satz von Assemblersprachen-Anweisungen übersetzt werden kann, beim Schritt **3510** vorhanden ist. Wenn eine derartige Vorschrift nicht vorhanden ist, kehrt das bevorzugte Verfahren auf den Schritt **3508** zurück, um eine andere RTL-Anweisung für ein Einbeziehen in die aktuelle RTL-Anweisungsgruppe auszuwählen. Wenn eine Vorschrift, die der aktuellen RTL-Anweisungsgruppe entspricht, existiert, erzeugt das hintere Compilerende beim Schritt **3512** einen Satz Assemblersprachen-Anweisungen entsprechend der Vorschrift. Nach dem Schritt **3512** stellt das hintere Compilerende fest, ob eine nächste RTL-Anweisung Beachtung im Kontext mit einer nächsten RTL-Anweisungsgruppe erfordert. wenn dem so ist, kehrt das bevorzugte Verfahren auf Schritt **3508** zurück; andernfalls ist das bevorzugte Verfahren beendet.

[0132] Die vorliegende Erfindung enthält vorzugsweise einen Compiler für ein dynamisch rekonfigurierbares Berechnen. In **Fig.** 11C und 11D ist ein Flußdiagramm von bevorzugten Compilieroperationen dargestellt, die von einem Compiler für ein dynamisch rekonfigurierbares Berechnen durchgeführt worden sind. Die bevorzugten Compileroperationen beginnen beim Schritt **3600** mit dem vorderen Ende des Compilers, der eine nächste Hochsprachen-Anweisung in einer Folge von Programmbefehlen auswählt. Als nächstes bestimmt das vordere Ende des Compilers beim Schritt **3602**, ob die ausgewählte Hochsprachen-Anweisung eine Rekonfigurationsanweisung ist. Wenn dem so ist, erzeugt das vordere Ende des Compilers eine RTL-Regkonfigurationsanweisung beim Schritt **3604**, worauf dann das bevorzugte Verfahren auf Schritt **3600** zurückkehrt. In der bevorzugten Ausführungsform ist die RTL-Rekonfigurationsanweisung eine nichtnormierte RTL-Anweisung, die einen ISA-Identifizierung erhält. Wenn beim Schritt **3602** die ausgewählte Hochprogramm-Anweisung nicht eine Rekonfigurationsanweisung ist, erzeugt das vordere Ende des Compilers als nächstes einen Satz RTL-Anweisungen in herkömmlicher Weise beim Schritt **3606**. Nach dem Schritt **3606** bestimmt das vordere Ende des Compilers beim Schritt **3608**, ob eine andere Hochsprachen-Anweisung Berücksichtigung erfordert. Wenn dem so ist, kehrt das bevorzugte Verfahren auf den Schritt **3600** zurück; anderenfalls geht das bevorzugte Verfahren auf Schritt **3610** über, um Operationen am Compilerende zu initiieren.

[0133] Beim Schritt **3610** führt das hintere Ende des Compilers für ein dynamisch rekonfigurierbares Berechnen Register-Zuordnungsoperationen durch. In der bevorzugten Ausführungsform der Erfindung ist jede ISA-Architektur so definiert, daß die Register-Architektur von einer IS-Architektur zur anderen folgerichtig ist; daher werden Register-Zuordnungsoperationen in herkömmlicher Weise durchgeführt. Der Fachmann erkennt, daß im allgemeinen eine folgerichtige Register-Architektur von einer ISA zur anderen keine absolute Forderung ist. Als nächstes wählt das hintere Ende des Compilers eine nächste RTL-Anweisung in einer aktuell in Betracht gezogenen RTL-Anweisungsgruppe beim Schritt **3612** aus. Das hintere Ende des Compilers bestimmt dann beim Schritt **3614**, ob die ausgewählte RTL-Anweisung eine RTL-Rekonfigurationsanweisung ist. Wenn die ausgewählte RTL-Anweisung nicht eine RTL-Rekonfigurationsanweisung ist, bestimmt das hintere Ende des Compilers beim Schritt **3618**, ob eine Vorschrift für die aktuell in Betracht gezogene RTL-Anweisungsgruppe existiert. Wenn dem nicht so ist, kehrt das bevorzugte Verfahren auf den Schritt **3612** zurück, um eine nächste RTL-Anweisung für ein Einbeziehen in die aktuell in Betracht gezogene RTL-Anweisungsgruppe zu wählen. In dem Fall, daß eine Vorschrift für die aktuell in Betracht gezogene RTL-Anweisungsgruppe beim Schritt **3618** existiert, erzeugt das hintere Ende des Compilers als nächstes einen Satz Assemblersprachen-Anweisungen beim Schritt **3620**, welche der aktuell in Betracht gezogenen RTL-Anweisungsgruppe gemäß dieser Vorschrift entsprechen. Nach dem Schritt **3620** bestimmt das hintere Ende des Compilers beim Schritt **3622**, ob eine andere RTL-Anweisung eine Berücksichtigung im Kontext mit einer nächsten RTL-Anweisungsgruppe erfordert. Wenn dem so ist, kehrt das bevorzugte Verfahren auf den Schritt **3612** zurück; anderenfalls endet das bevorzugte Verfahren.

[0134] Wenn beim Schritt **3614** die ausgewählte RTL-Anweisung eine RTL-Rekonfigurationsanweisung ist, wählt das hintere Ende des Compilers einen Vorschriftensatz. beim Schritt **3616** aus, welcher der ISA-Identifizierung der RTL-Rekonfigurationsanweisung entspricht. In der vorliegenden Erfindung existiert vorzugsweise ein eindeutiger Vorschriftensatz für jede ISA. Jeder Vorschriftensatz schafft daher eine oder mehrere Vorschrif-

ten, um Gruppen von RTL-Anweisungen in Assemblersprachen-Anweisungen entsprechend einer ganz bestimmten IS-Architektur umzuwandeln. Nach dem Schritt **3616** geht das bevorzugte Verfahren auf Schritt **3618** über. Der Vorschriftensatz, der einer vorgegebenen IS-Architektur entspricht, enthält vorzugsweise ein Vorschrift, um die RTL-Rekonfigurationsanweisung in einen Satz Assemblersprachen-Befehle zu übersetzen, die eine Software-Unterbrechnung erzeugen, die auf eine Durchführung eines Rekonfigurations-Abwicklers (handler) hinausläuft, wie im einzelnen unten beschrieben wird.

[0135] In der vorstehend beschriebenen Weise erzeugt der Compiler für dynamisch rekonfigurierbares Berechnen selektiv und automatisch Assemblersprachen-Anweisungen entsprechend mehreren IS-Architekturen während Compileroperationen. Mit anderen Worten, während des Compilerprozesses übersetzt der Compiler einen einzigen Satz Programmbefehlen entsprechend einer variablen IS-Architektur. Der Compiler ist vorzugsweise ein herkömmlicher Compiler, der modifiziert worden ist, um bevorzugte Compileroperationen durchzuführen, die vorstehend unter Bezugnahme auf **Fig.** 11C und 11D beschrieben worden sind. Der Fachmann erkennt, daß, obwohl die geforderten Modifikationen nicht komplex sind, solche Modifikationen im Hinblick sowohl auf herkömmliche Compilertechniken als auch im Hinblick auf herkömmliche rekonfigurierbaren Berechnungsmethoden nicht offensichtlich und naheliegend sind.

[0136] Die Lehren der vorliegenden Erfindung unterscheiden sich deutlich von anderen Systemen und Verfahren für ein umprograinmierbares oder rekonfigurierbares Rechnen. Insbesondere ist die vorliegende Erfindung nicht äquivalent mit einer herunterladbaren Mikrocode-Architektur, da solche Architekturen im allgemeinen auf nicht-rekonfigurierbare Steuereinrichtung und eine nicht-rekonfigurierbare Hardware angewiesen sind. Die vorliegende Erfindung unterscheidet sich also deutlich von einem angeschlossen rekonfiguierbaren Prozessor-(ARP-)System, in welchem eine Gruppe von rekonfigurierbaren Hardware-Ressourcen mit einem nicht-rekonfigurierbaren Host-Prozessor oder Host-System verbunden ist. Die ARP-Einrichtung hängt von dem Host ab, um gewisse Programmbefehle durchzuführen. Daher wird eine Menge verfügbarer Silizium-Ressourcen nicht maximal über den Zeitrahmen der Programmdurchführung genutzt, da Silizium-Ressourcen bei der ARP-Einrichtung oder dem Host unbenutzt sind oder ineffizient genutzt werden, wenn der Host bzw. die ARP-Einrichtung mit Daten arbeitet. Im Unterschied hierzu ist jede S-Einrichtung **3012** ein unabhängiger Rechner, in welchem ganze Programme ohne weiteres ausgeführt werden können. Mehrere S-Einrichtungen **3012** führen vorzugsweise gleichzeitig Programme durch. Die vorliegende Erfindung lehrt daher die ständige maximale Ausnutzung von Silizium-Ressourcen sowohl für einzelne Programme, die von einzelnen S-Einrichtungen **3012** durchgeführt werden oder von mehreren Programmen, die von dem gesamten System **3010** ausgeführt werden.

[0137] Eine ARP-Einrichtung stellt einen Rechenbeschleuniger für einen ganz bestimmten Algorithmus in einer ganz bestimmten Zeit zur Verfügung und ist als ein Satz von Verknüpfungsgliedern ausgeführt, die optimal bezüglich dieses spezifischen Algorithmus miteinander verbunden sind. Die Verwendung von rekonfigurierbaren Hardware-Ressourcen für universelle Operationen, wie eine verwaltende Befehlsausführung, ist bei ARP-System vermieden. Darüber hinaus behandelt ein ARP-System nicht eine vorgegebene Menge von miteinander verbundenen Verknüpfungsgliedern bzw. Gates als eine ohne weiteres wiederverwendbare Ressource. Im Gegensatz, die vorliegende Erfindung lehrt eine dynamisch rekonfigurierbare Verarbeitungseinrichtung, die für ein effizientes Management einer Befehlsausführung gemäß einem Befehlsausführungsmodells konfiguriert ist, das am besten für die Rechenerfordernisse zu einem ganz bestimmten Zeitpunkt ausgelegt ist. Jede S-Einrichtung **3012** weist eine Vielzahl ohne weiteres wiederverwendbarer Ressourcen, beispielsweise das ISS **3100**, die Unterbrechungslogik **3106** und die Speicher/Ausrichtlogik **3152** auf. Die vorliegende Erfindung lehrt die Verwendung von rekonfigurierbaren logischen Ressourcen auf dem Niveau von LCBs- oder IOBs-Gruppen und rekonfigurierbarer Verbindungen, jedoch nicht auf dem Niveau von miteinander verbundenen Gates. Die vorliegende Erfindung lehrt folglich die Verwendung von rekonfigurierbaren höherwertigen logischen Designkonstrakts, die zum Durchführen von Operationen der ganzen Klassen von Rechenproblemen verwendbar sind, und lehrt nicht ein brauchbares Verbindungsschema, das für einen einzigen Algorithmus verwendbar ist.

[0138] Im allgemeinen sind ARP-Systeme auf ein Übertragen eines ganz bestimmten Algorithmus in einen Satz von miteinander verbundenen Gates gerichtet. Einige ARP-Systeme versuchen, hochwertige Befehle in einer optimalen Hardware-Konfiguration zu compilieren, welches im allgemeinen ein hartes NP-Problem ist. Im Unterschied hierzu lehrt die Erfindung die Verwendung eines Compilers für ein dynamisch rekonfigurierbares Berechnen, das hochwertige Programmbefehle in Assembler-Sprachenbefehle gemäß einer variablen ISA auf sehr unkomplizierte Weise compiliert.

[0139] Eine ARP-Einrichtung ist im allgemeinen nicht in der Lage, ihre eigenes Host-Programm als Daten zu

behandeln oder es selbst zu kontextualisieren. Im Unterscheid hierzu kann jede S-Einrichtung in dem System **3010** ihre eigenen Programme als Daten behandeln, und folglich ohne weiteres selbst kontextualisieren. Das System **3010** kann ohne weiteres sich selbst durch die Ausführung seiner eigenen Programme simmulieren. Die vorliegende Erfindung hat zusätzlich die Fähigkeit, ihren eigenen Compiler zu compilieren.

[0140] In der vorliegenden Erfindung kann ein einziges Programm eine erste Gruppe von Befehlen, die zu einem ersten ISA gehören, eine zweite Gruppe von Befehlen, die zu einer zweiten ISA gehören, eine dritte Gruppe von Befehlen, die zu noch einer weiteren ISA gehören, usw. enthalten. Die hier beschriebene Architektur führt jede derartige Gruppe von Befehlen mit Hilfe von Hardware durch, die hinsichtlich Durchlaufzeit konfiguriert ist, um die ISA durchzuführen, zu welcher die Befehle gehören. Keine bekannten Systeme oder Methoden bieten ähnliche Lehren an.

[0141] Die Erfindung lehrt ferner ein rekonfigurierbares Unterbrechungsschema, bei welchem Unterbrechungslatenz, Unterbrechungspräzision und ein programmierbares Zustandsübergangs-Freigeben gemäß der aktuellen, in Betracht gezogenen ISA sich ändern kann. Keine anlogen Lehren werden in anderen Computersystemen gefunden. Die vorliegende Erfindung lehrt zusätzlich ein Computersystem mit einer rekonfigurierbaren Datenweg-Bitbreite, einer Adressen-Bitbreite und rekonfigurierbare Steuerzeilen-Breiten im Unterschied zu herkömmlichen Computersystemen.

[0142] Zusammenfassend wurde ein Kompiliersystem und ein Verfahren zur Erzeugung einer Folge von Programmbefehlen zur Verwendung in einer dynamisch rekonfigurierbaren Verarbeitungseinheit geschaffen, die eine interne Hardwareorganisation aufweist, die wahlweise unter einer Anzahl von Hardwarearchitekturen geändert werden kann, wobei jede Hardwarearchitektur Befehle von einem entsprechenden Befehlsatz ausführt. Quelldateien werden zur Ausführung mit Hilfe von mehreren Befehlsatzarchitekturen (instruction set architectures) kompiliert, wie dies durch Rekonfigurations-Übersetzungsanweisungen spezifiziert wird. Die Objektdateien fassen wahlweise Bitströme, die Hardwarearchitekturen spezifizieren, die Befehlsatzarchitekturen entsprechen, mit ausführbarem Code zur Ausführung auf den Architekturen zusammen.

Bezugszeichenliste

Fig. 1

12 – 5-Einrichtung
14 – T-Einrichtung
18 – Ein/Ausgabe-T-Einrichtung
20 – Ein/Ausgabeeinrichtung
16 – Mehrzweckverbindungsmatrix (GPI-Matrix)
22 – Master-Zeitbasiseinheit

Fig. 1A

131 – Taktgenerator
132 – Bitstromspeicher
133 – Programm-/Datenspeicher
12 – S-Einrichtung
14 – T-Einrichtung

Fig. 1B

149 – Speicherbus
140 – FPGA Konfigurationshardware
146 – Befehle dekodieren
147 – Speicherinterface
132 – ISA0 Bitstrom;
ISA1 Bitstrom

Fig. 1C

siehe **Fig.** 1B

Fig. 3

301 – Quelldatei lesen
302 – ISA identifizieren
303 – Rekonfigurationsanweisung erzeugen
304 – Anweisungen für identifizierte ISA kompilieren
305 – weitere ISA?
306 – übersetzen
307 – binden
308 – laden
309 – Ende

Fig. 3A

600 – nächste Hochsprachenanweisung auswählen
601 – Funktionsaufruf
602 – verschiedene ISA?
603 – RTL-Code abgeben
605 – RTL-Funktionsaufruf abgeben
607 – RTL-Code zum Speichern lebender Register abgeben
604 – RTL-Rekonfigurationscode abgeben
606 – RTL-Funktionsaufruf abgeben
608 – weitere Hochsprachenanweisung?
609 – RTL-Rekonfigurationscode abgeben
611 – RTL-Code zum Wiederherstellen lebender Register abgeben
613 – RTL-Code zum Lesen von Rückgabewert abgeben

Fig. 3B

612 – nächste RTL-Anweisung auswählen
618 – Regel für aktuelle RTL-Anweisungsgruppe erhalten
620 – Maschinensprachenanweisung erzeugen. Gemäß der Regel für diese ISA setzen
622 – andere RTL-Anweisung?
610 – Registerreservierung ausführen

Fig. 3C

331 – RTL-Code mit neuer ISA mit Bemerkungen versehen
332 – ISA-abhängige und ISA-unabhängige Optimierung
333 – maschinenabhängige Anweisungen erzeugen

Fig. 4

401 – Quelle
403 – Objekt
404 – Binder
405 – ausführbares Programm
406 – ISA-Bitströme
10 – rekonfigurierbarer Computer
407 – Ladeprogramm

Fig. 5

501 – Binden-Ansicht
503 – ELF-Kopfteil
504 – Programmkopfteiltabelle optional
505 – Abschnitt 1
...
Abschnitt n
...

...
506 – Abschnittskopfteiltabelle
502 – Ausführen-Ansicht
503 – ELF-Kopfteil
504 – Programmkopfteiltabelle
505 – Abschnitt 1
Abschnitt 2
...
506 – Abschnittskopfteiltabelle optional

Fig. 6

651 – Anweisung reconfig
654 – vorbestimmte Adresse < = SP
655 – Hardware lädt nächste Konfiguration
656 – SP < = vorbestimmte Adresse

Fig. 7

707 – lebende Registerwerte retten
701 – Parameter zu aufgerufener Funktion schieben
702 – rekonfigurieren zu neuer ISA
703 – Unterprogramm aufrufen
704 – Fluß zum Aufrufer zurückkehren
705 – auf ursprüngliche ISA rekonfigurieren
706 – Rückgabewert lesen
708 – lebende Registerwerte wieder herstellen

Patentansprüche

1. Kompilierverfahren mittels eines Kompilers zur Erzeugung einer Folge (**50**) von Programmbefehlen und Rekonfigurations-Anweisungen zur Ausführung in einem dynamisch rekonfigurierbaren Computer (**10**), der ein Prozessor-Modul (**130**) aufweist, das ein dynamisch rekonfigurierbares Prozessor-Submodul (**12**), einen mit dem Prozessor-Submodul (**12**) verbundenen Programm/Daten-Speicher (**133**) und einen mit dem Prozessor-Submodul (**12**) verbundenen Bitstrom-Speicher (**132**) umfasst, wobei das Prozessor-Submodul durch Laden eines Bitstroms aus dem Bitstrom-Speicher auf Rekonfigurations-Anweisungen hin während der Ausführung der Folge von Programm-Befehlen wahlweise unter einer Anzahl von Befehlssatz-Architekturen (ISA) rekonfiguriert werden kann, mit den folgenden Schritten:
a) als Eingabe wird eine Quelldatei (**301**) empfangen, die eine Anzahl von Quellcode-Befehlsanweisungen enthält, und zwar einschließlich mindestens eines ersten Untersatzes von Quellcode-Befehlsanweisungen und eines zweiten Untersatzes von Quellcode-Befehlsanweisungen;
b) für den ersten Untersatz von Quellcode-Befehlsanweisungen wird ein erster Befehlsatz, der einer ersten Befehlsatz-Architektur entspricht, mittels einer ersten im Quellcode enthaltenen Rekonfigurations-Übersetzungsanweisung identifiziert, wobei die erste Rekonfigurations-Übersetzungsanweisung den ersten Befehlsatz spezifiziert;
c) für den zweiten Untersatz von Quellcode-Befehlsanweisungen wird ein zweiter Befehlsatz, der einer zweiten Befehlsatz-Architektur entspricht, mittels einer zweiten im Quellcode enthaltenen Rekonfigurations-Übersetzungsanweisung identifiziert, wobei die zweite Rekonfigurations-Übersetzungsanweisung den zweiten Befehlsatz spezifiziert; und
d) der erste Untersatz von Quellcode-Befehlsanweisungen wird unter Verwendung des ersten Befehlsatzes kompiliert und der zweite Untersatz von Quellcode-Befehlsanweisungen wird unter Verwendung des zweiten Befehlsatzes kompiliert;
e) wobei für eine als mit dem ersten Befehlsatz gekennzeichnete Quellcode-Befehlsanweisung (**600**) nur kompilierte Programm-Befehle (**603**, **605**) erzeugt werden, falls die ausgewählte Quellcode-Befehlsanweisung keinen Funktionsaufruf (**601**) oder einen Funktionsaufruf zu einer ebenfalls als mit dem ersten Befehlsatz gekennzeichneten Funktion enthält;
f) wobei in Reaktion auf eine als mit dem ersten Befehlsatz gekennzeichnete Quellcode-Befehlsanweisung (**600**), die (**601**) einen Funktionsaufruf zu einer als mit dem zweiten Befehlsatz gekennzeichneten (**602**) Funktion enthält, die folgenden Schritte der Reihe nach ausgeführt werden:
f.1) (**607**) ein Zustand der Programm-Ausführung wird in einem vorbestimmten Speicherbereich gespeichert,

f.2) (**604**) Rekonfigurations-Anweisungen, die erforderlich sind, um eine Rekonfiguration von der ersten Befehlsatz-Architektur zur zweiten Befehlsatz-Architektur zu erzielen, werden an das Prozessor-Submodul abgegeben;
f.3) (**606**) nach der Rekonfiguration wird für den Funktionsaufruf ein kompilierter Programm-Befehl in der zweiten Befehlsatz-Architektur abgegeben;
f.4) (**609**) nach Ablauf der Funktion wird eine Rekonfigurations-Anweisung zur Rekonfiguration des Prozessor-Submodules in den ersten Befehlsatz abgegeben und f.5) (**602**) der Zustand der Programm-Ausführung zu Beginn der Rekonfiguration wird durch Laden aus dem vorbestimmten Speicherbereich wieder hergestellt.
g) die Schritte e) und f) werden für jede Quellcode-Befehlsanweisung in der Quelldatei wiederholt.

2. Verfahren nach Anspruch 1, bei dem jede Rekonfigurations-Übersetzungsanweisung unter Verwendung einer Meta-Syntax bereitgestellt wird.

3. Verfahren nach Anspruch 1 oder 2, bei dem jede Rekonfigurations-Übersetzungsanweisung entweder eine Direkt-Rekonfigurations-Übersetzungsanweisung, eine Funktionsebenen-Rekonfigurations-Übersetzungsanweisung oder eine Standard-Rekonfigurations-Übersetzungsanweisung umfasst.

4. Verfahren nach einem der vorhergehenden Ansprüche, mit dem weiteren Schritt:
es wird eine ausführbare Datei erzeugt, die die Ergebnisse der Kompilation beinhalten und außerdem für jeden Untersatz von Befehlsanweisungen einen Rekonfigurationscode, der den Befehlsatz identifiziert, der dem Untersatz von Befehlsanweisungen entspricht.

5. Verfahren nach einem der Ansprüche 1 bis 4, mit dem weiteren Schritt:
es wird eine ausführbare Datei erzeugt, die die Ergebnisse des Kompilierens beinhaltet und außerdem für jeden Untersatz von Befehlsanweisungen einen Verweis, der einen Bitstrom bestimmt, der den Befehlsatz darstellt, der dem Untersatz von Befehlsanweisungen entspricht.

6. Verfahren nach einem der Ansprüche 1 bis 4, mit dem weiteren Schritt:
es wird eine ausführbare Datei erzeugt, die die Ergebnisse des Kompilierens beinhaltet und außerdem für den Untersatz von Befehlsanweisungen einen Verweis, der entsprechend einem erweiterten, ausführbaren Programm und einem Bindungsformat codiert ist, wobei der Verweis einen Bistrom bestimmt, der den Befehlsatz darstellt, der dem Untersatz von Befehlsanweisungen entspricht.

7. Verfahren nach einem der Ansprüche 1 bis 4, mit dem weiteren Schritt:
es wird eine ausführbare Datei erzeugt, die die Ergebnisse des Kompilierens beinhaltet und außerdem für jeden Untersatz von Befehlsanweisungen einen Bistrom, der den Befehlsatz darstellt, der dem Untersatz von Befehlsanweisungen entspricht.

8. Verfahren nach einem der Ansprüche 1 bis 4, mit den weiteren Schritten:
h) eine erste Objektdatei wird erzeugt, die die Ergebnisse des Kompilierens beinhaltet und außerdem für jeden Untersatz von Befehlsanweisungen einen Rekonfigurationscode, der den Befehlsatz identifiziert, der dem Untersatz von Befehlsanweisungen entspricht;
i) die Schritte a) bis h) werden mindestens für eine zweite Quelldatei wiederholt, um mindestens eine zweite Objektdatei zu erzeugen; und
j) die in den Schritten h) und i) erzeugten Objektdateien werden gebunden, um eine ausführbare Datei zu erzeugen.

9. Verfahren nach Anspruch 8, mit dem weiteren Schritt:
k) an der erzeugten, ausführbaren Datei wird entsprechend den Ausrichtungserfordernissen eine Speicherausrichtung vorgenommen.

10. Verfahren nach Anspruch 9, bei dem die erzeugte, ausführbare Datei einem Bitstrom zugeordnet ist, der einen Befehlsatz darstellt, und bei dem der Schritt k) den Schritt umfasst:
k.1) der Bitstrom wird aufgefüllt, um eine Speicherausrichtung durchzuführen.

11. Verfahren nach Anspruch 8, bei dem:
beim Schritt a) als Eingabe eine Quelldatei empfangen wird, die eine Anzahl von Quellcode-Befehlsanweisungen einschließlich mindestens eines ersten Untersatzes von Befehlsanweisungen enthält, wobei zumindest eine der Befehlsanweisungen einen externen Verweis enthält; und
beim Schritt h) eine erste Objektdatei erzeugt wird, die die Ergebnisse des Kompilierens enthält und außerdem

für jeden Untersatz von Befehlsanweisungen einen Rekonfigurationscode, der den Befehlsatz identifiziert, der dem Untersatz von Befehlsanweisungen entspricht, wobei zumindest eine der Befehlsanweisungen einen externen Verweis enthält; und
mit dem weiteren Schritt:
i.1) vor der Ausführung des Schrittes j) werden für jede Objektdatei die externen Verweise aufgelöst.

12. Verfahren nach einem der vorhergehenden Ansprüche, bei dem der erste Untersatz von Befehlsanweisungen eine erste definierte Funktion und der zweite Untersatz von Befehlsanweisungen eine zweite definierte Funktion umfasst.

13. Verfahren nach einem der Ansprüche 1 bis 12, bei dem der erste Untersatz von Befehlsanweisungen einen ersten beliebigen Block von Anweisungen und der zweite Untersatz von Befehlsanweisungen einen zweiten beliebigen Block von Anweisungen umfasst.

14. Verfahren nach Anspruch 13, bei dem die Quelldatei mindestens einen Funktionsaufruf und einen Funktionsrücksprung umfasst und bei dem die Schritte b) und c) jeweils die wahlweise Ausführung einer interprozeduralen Analyse umfassen, um bei jedem Funktionsaufruf und bei jedem Funktionsrücksprung eine im Kontext befindliche Befehlsatzarchitektur (ISA) zu identifizieren.

15. Verfahren nach Anspruch 1, mit den weiteren Schritten:
h) der erste kompilierte Untersatz von Befehlsanweisungen wird für den ersten Befehlsatz optimiert; und
i) der zweite kompilierte Untersatz von Befehlsanweisungen wird für den zweiten Befehlsatz optimiert.

16. Verfahren nach Anspruch 1, bei dem der Schritt f.1) den folgenden Schritt umfasst:
es wird eine Codeanweisung zum Retten von lebenden Registern abgegeben; und bei dem der Schritt f.5) den folgenden Schritt umfasst:
es wird eine Codeanweisung zur Wiederherstellung der geretteten, lebenden Register abgegeben.

17. Verfahren nach Anspruch 16, bei dem die Codeanweisungen Registertransferebenenanweisungen (RTL-Anweisungen) umfassen.

18. Verfahren nach Anspruch 17, mit den weiteren Schritten:
eine Registerreservierung wird durchgeführt;
für jede Registertransferebenenanweisung wird:
bestimmt, ob für die Registertransferebenenanweisung eine Übersetzungsregel existiert; und
in Antwort auf die Feststellung, dass eine Übersetzungsregel existiert, wird für die Registertransferebenenanweisung entsprechend der Übersetzungsregel ein Assemblercode erzeugt.

19. Verfahren nach Anspruch 17, mit den weiteren Schritten:
jede Registertransferebenenanweisung wird mit Bemerkungen versehen, um eine Befehlsatzarchitektur (ISA) anzugeben;
die Registertransferebenenanweisungen werden optimiert; und
aus den optimierten Registertransferebenenanweisungen wird ein maschinenabhängiger Assemblercode erzeugt.

20. Verfahren nach einem der Ansprüche 1 bis 19, bei dem der Schritt f.1) die folgenden Schritte umfasst:
Zustandsvariablen werden auf einem Stapel gespeichert, auf den mit Hilfe eines Stapelzeigers (SP) verwiesen wird; und
der Stapelzeiger wird in einer Speicherstelle abgespeichert; und
bei dem der Schritt f.5) die Schritte umfasst:
der Stapelzeiger wird von der Speicherstelle wieder abgerufen; und
die Zustandvariablen werden von dem Stapel wieder abgerufen.

21. Computerprogramm zur Erzeugung einer Folge von Programmbefehlen zur Ausführung in einer dynamisch rekonfigurierbaren Verarbeitungseinheit, dadurch gekennzeichnet, dass das Computerprogramm das Verfahren nach einem der Ansprüche 1 bis 21 ausführt.

22. Computerverwendbares Datenspeichermedium, auf dem die Programmbefehle des Computerprogramms nach Anspruch 21 gespeichert sind.

Es folgen 20 Blatt Zeichnungen

Anhängende Zeichnungen

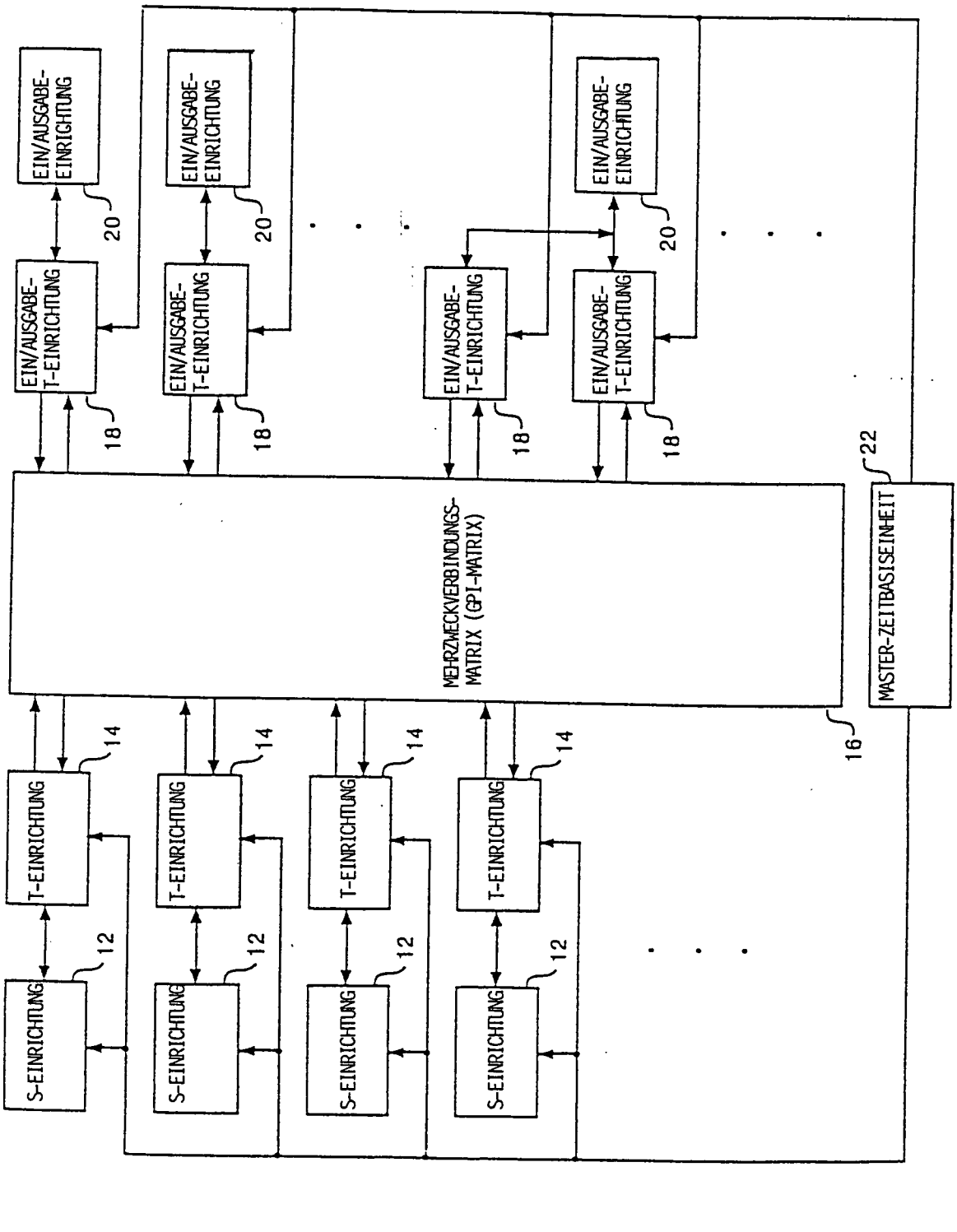
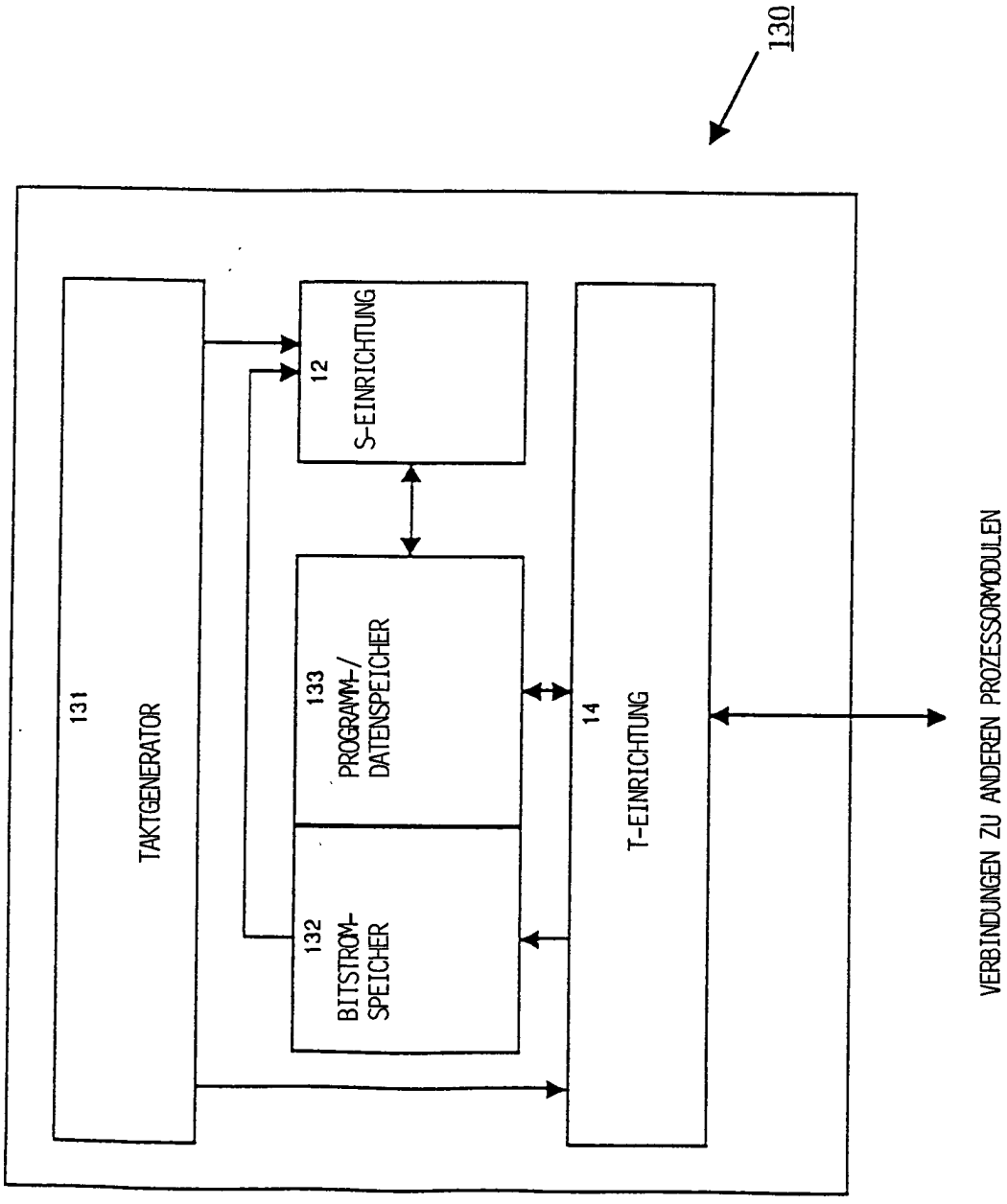


FIG. 1


130
131
TAKTGENERATOR
12
S-EINRICHTUNG
133
PROGRAMM-/
DATENSPEICHER
132
BITSTROM-
SPEICHER
14
T-EINRICHTUNG
VERBINDUNGEN ZU ANDEREN PROZESSORMODULEN

FIGUR 1A

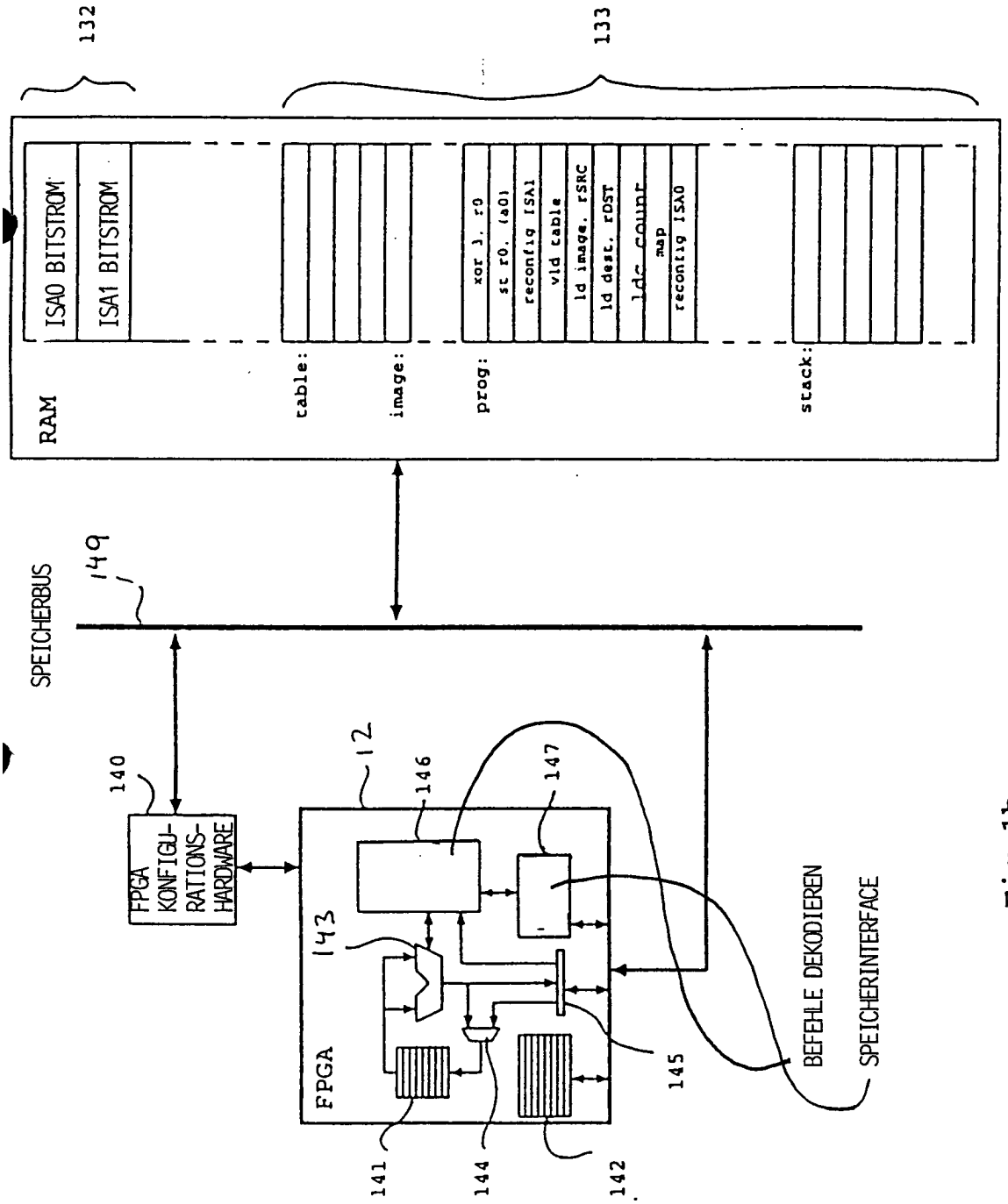

132
133
RAM
ISA0 BITSTROM
ISA1 BITSTROM
cable:
image:
prog:
stack:
xor 1, r0
sc r0, (a0)
reconfig ISA1
vld cable
ld image, rSRC
ld dest, rDST
ldc count
map
reconfig ISA0
SPEICHERBUS
149
140
FPGA KONFIGU-RATIONS-HARDWARE
12
146
147
143
FPGA
145
141
144
142
BEFEHLE DEKODIEREN
SPEICHERINTERFACE

Fig 1b

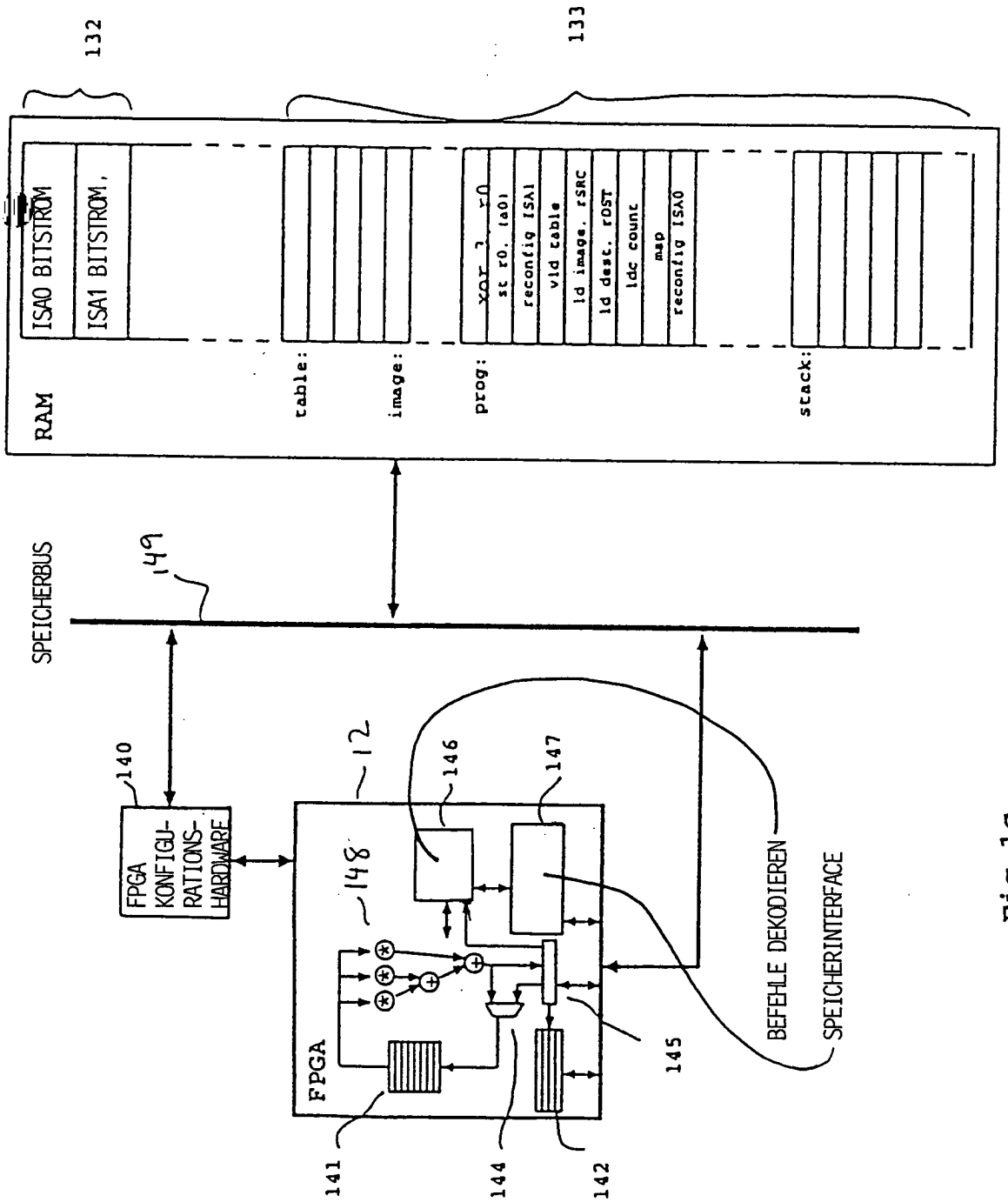
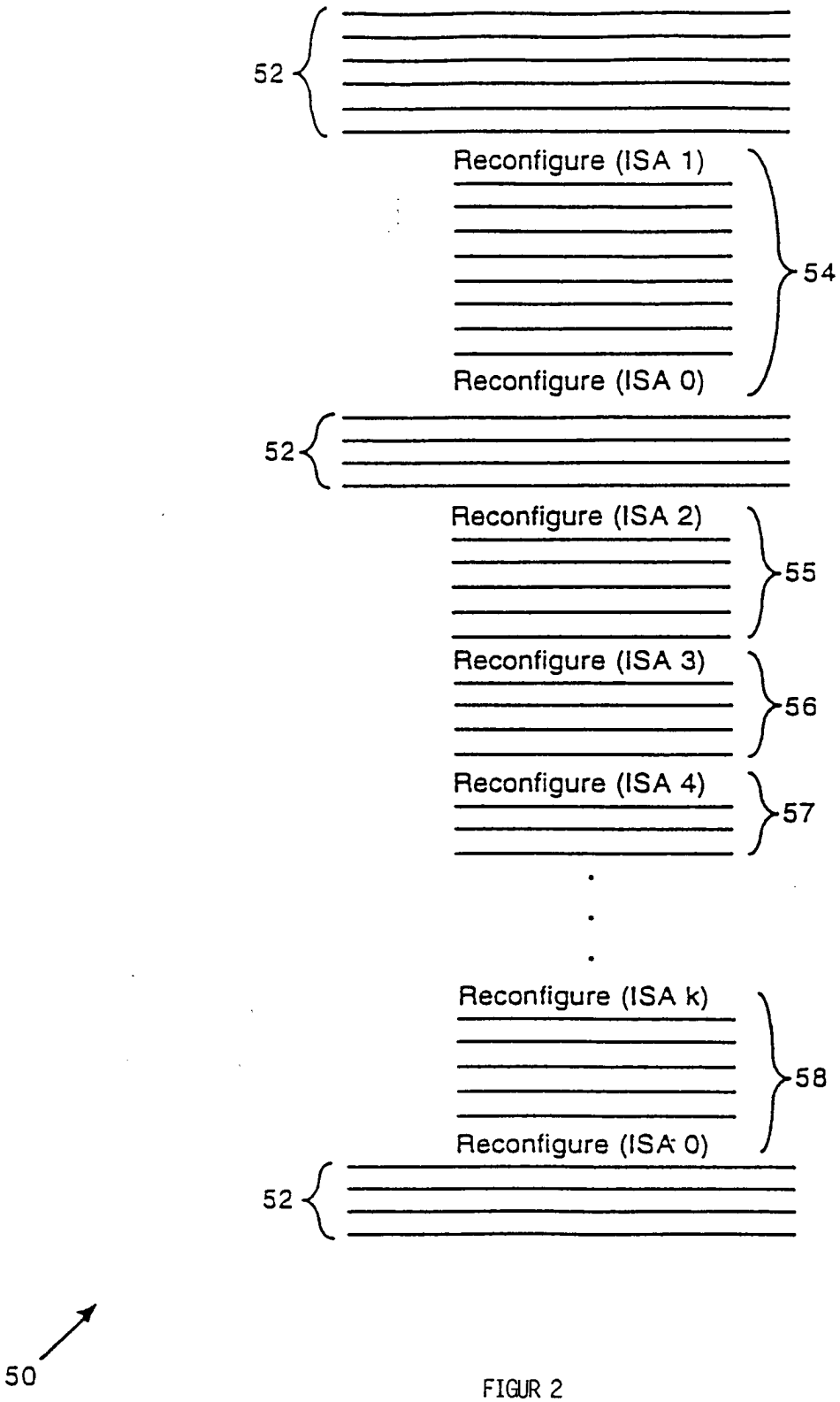
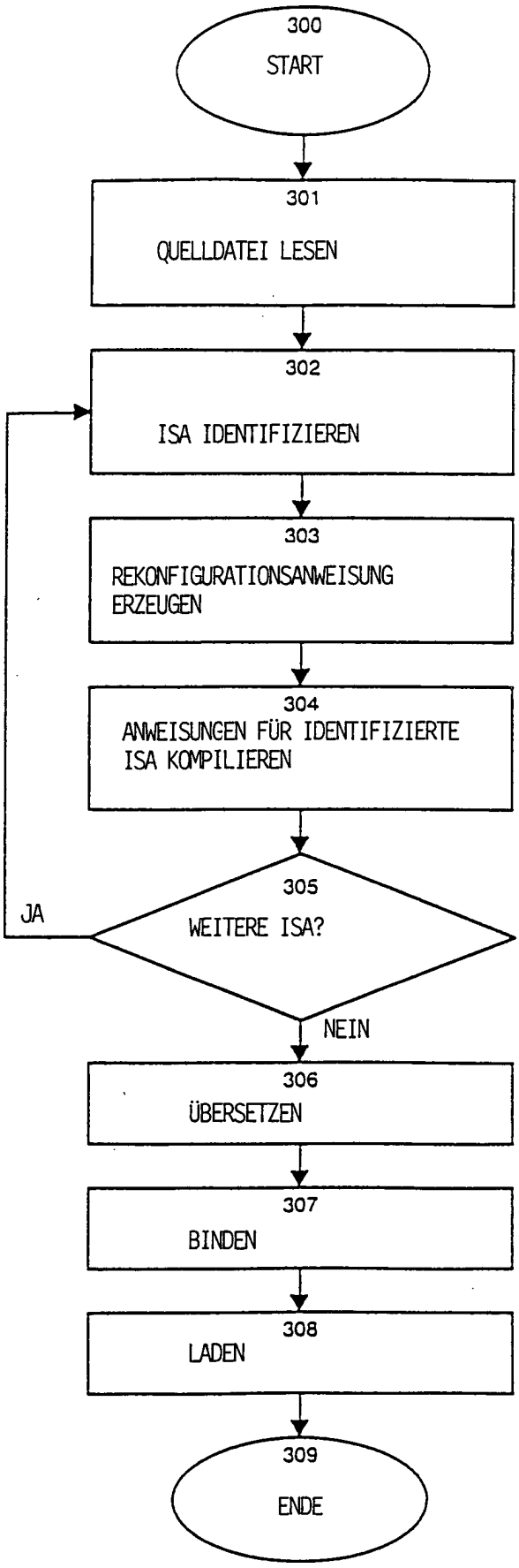

132
133
RAM
ISA0 BITSTROM
ISA1 BITSTROM,
table:
image:
prog:
stack:
xor r, r0
st r0, (a0)
reconfig ISA1
vld table
ld image, rSRC
ld dest, rDST
ldc count
map
reconfig ISA0
SPEICHERBUS
149
140
FPGA KONFIGU-RATIONS-HARDWARE
12
146
147
148
FPGA
145
BEFEHLE DEKODIEREN
SPEICHERINTERFACE
141
144
142

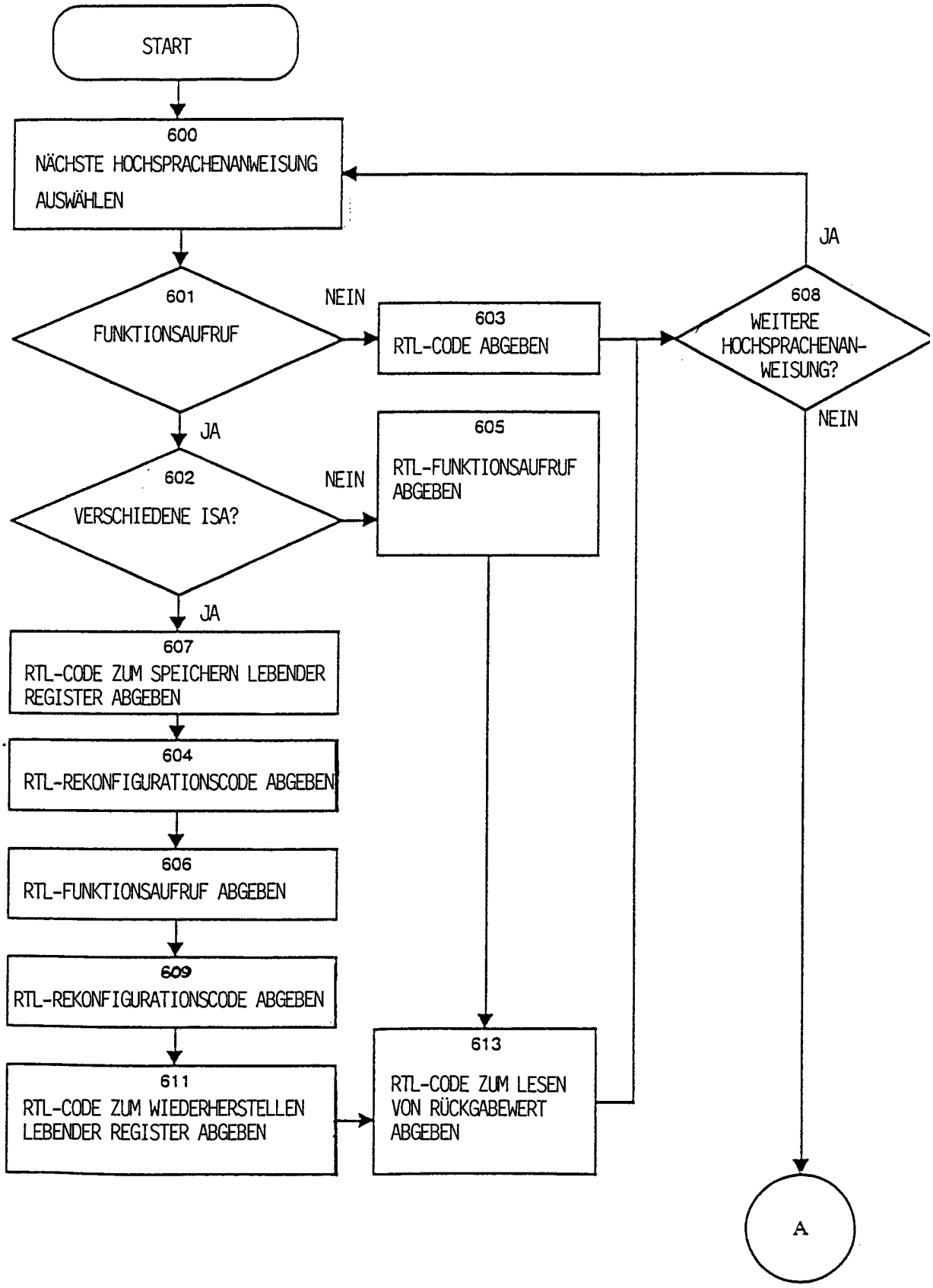
Fig 1c


52
Reconfigure (ISA 1)
54
Reconfigure (ISA 0)
52
Reconfigure (ISA 2)
55
Reconfigure (ISA 3)
56
Reconfigure (ISA 4)
57
Reconfigure (ISA k)
58
Reconfigure (ISA 0)
52
50

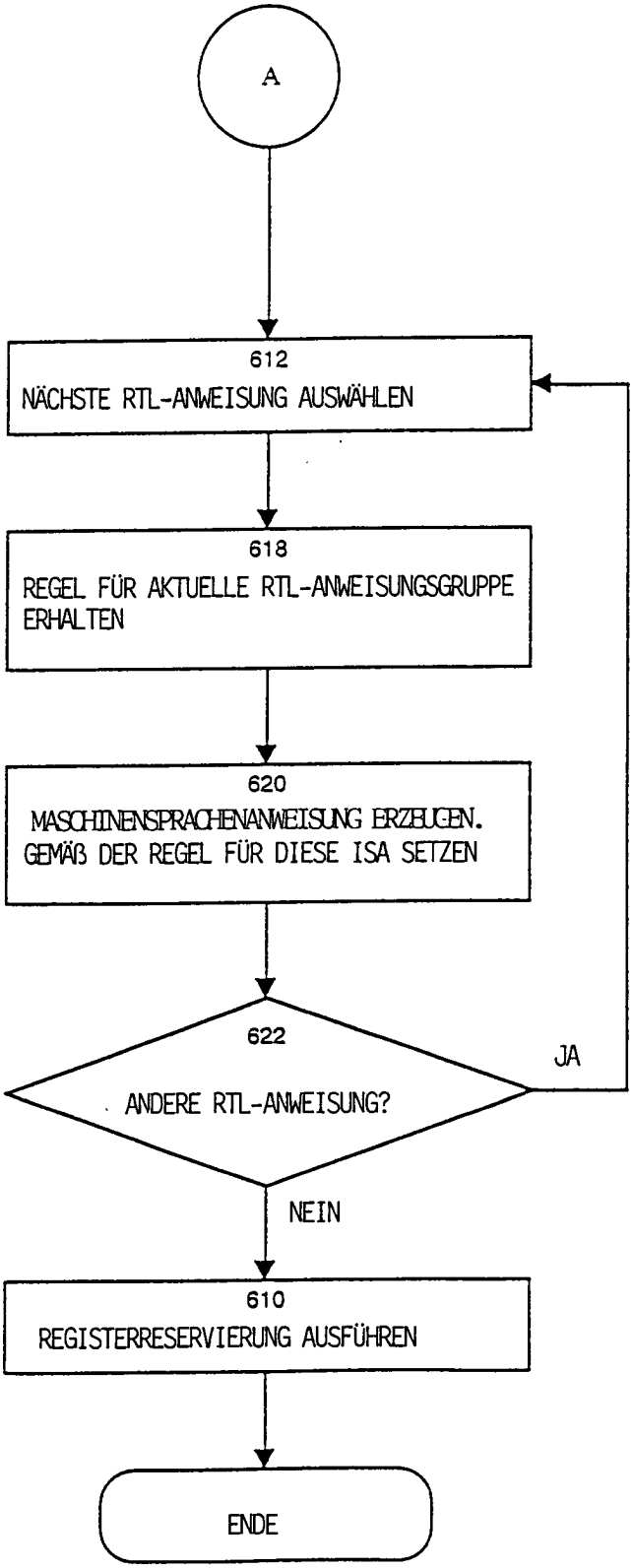
FIGUR 2


300
START
301
QUELLDATEI LESEN
302
ISA IDENTIFIZIEREN
303
REKONFIGURATIONSANWEISUNG ERZEUGEN
304
ANWEISUNGEN FÜR IDENTIFIZIERTE ISA KOMPILIEREN
305
WEITERE ISA?
JA
NEIN
306
ÜBERSETZEN
307
BINDEN
308
LADEN
309
ENDE

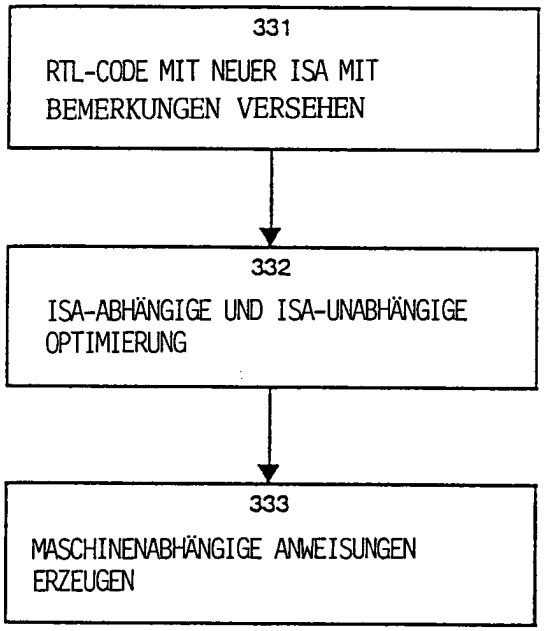
FIGUR 3


START
600
NÄCHSTE HOCHSPRACHENANWEISUNG AUSWÄHLEN
601
FUNKTIONSAUFRUF
NEIN
603
RTL-CODE ABGEBEN
608
WEITERE HOCHSPRACHENAN-WEISUNG?
JA
NEIN
JA
602
VERSCHIEDENE ISA?
NEIN
605
RTL-FUNKTIONSAUFRUF ABGEBEN
JA
607
RTL-CODE ZUM SPEICHERN LEBENDER REGISTER ABGEBEN
604
RTL-REKONFIGURATIONSCODE ABGEBEN
606
RTL-FUNKTIONSAUFRUF ABGEBEN
609
RTL-REKONFIGURATIONSCODE ABGEBEN
611
RTL-CODE ZUM WIEDERHERSTELLEN LEBENDER REGISTER ABGEBEN
613
RTL-CODE ZUM LESEN VON RÜCKGABEWERT ABGEBEN
A

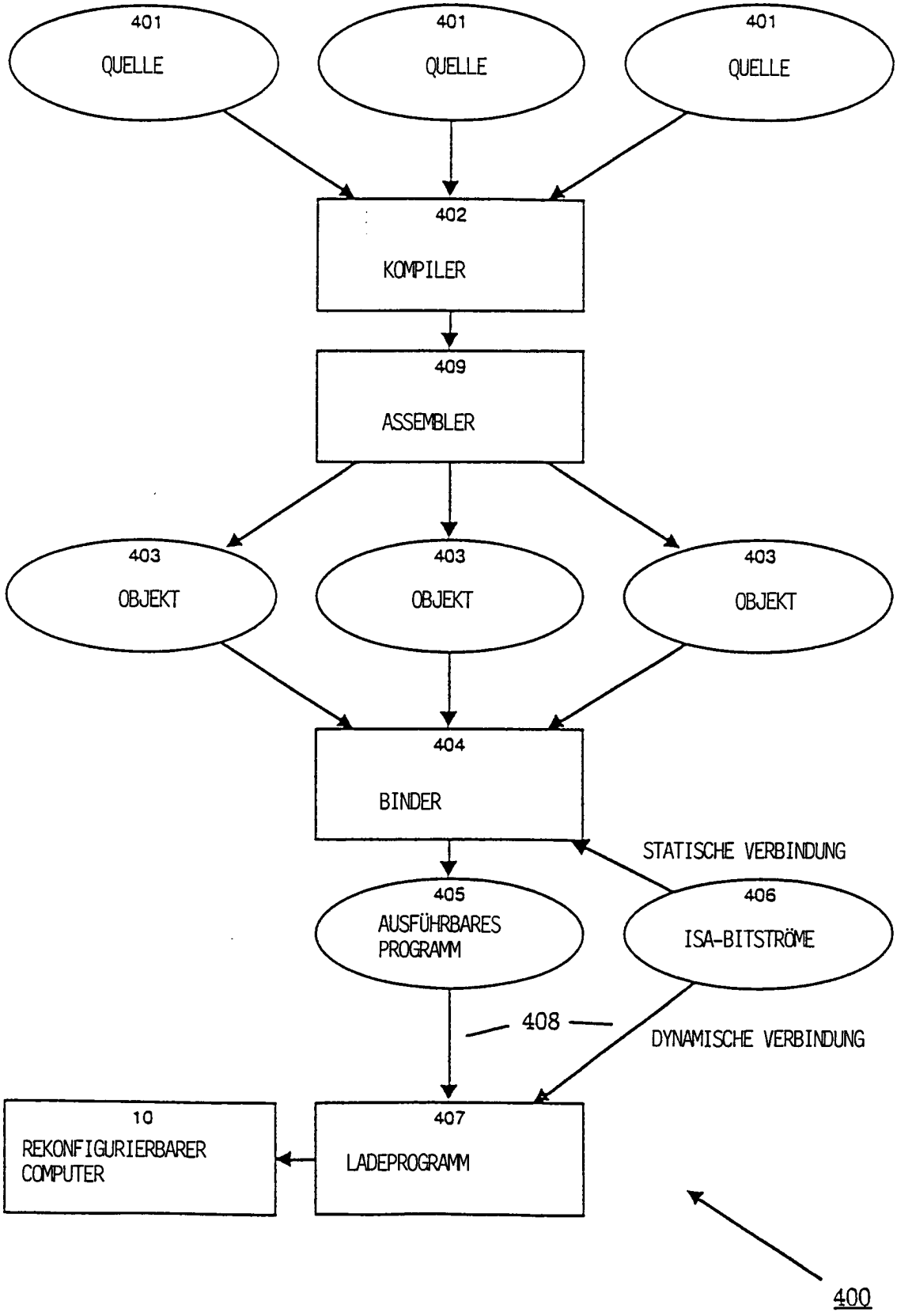
FIGUR 3A


A
612
NÄCHSTE RTL-ANWEISUNG AUSWÄHLEN
618
REGEL FÜR AKTUELLE RTL-ANWEISUNGSGRUPPE ERHALTEN
620
MASCHINENSPRACHENANWEISUNG ERZEUGEN. GEMÄß DER REGEL FÜR DIESE ISA SETZEN
622
ANDERE RTL-ANWEISUNG?
JA
NEIN
610
REGISTERRESERVIERUNG AUSFÜHREN
ENDE

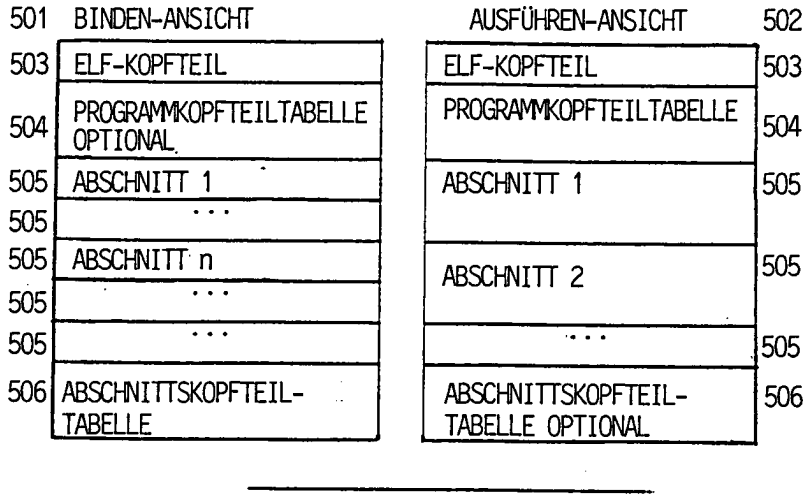
FIGUR 3B


331
RTL-CODE MIT NEUER ISA MIT BEMERKUNGEN VERSEHEN
332
ISA-ABHÄNGIGE UND ISA-UNABHÄNGIGE OPTIMIERUNG
333
MASCHINENABHÄNGIGE ANWEISUNGEN ERZEUGEN

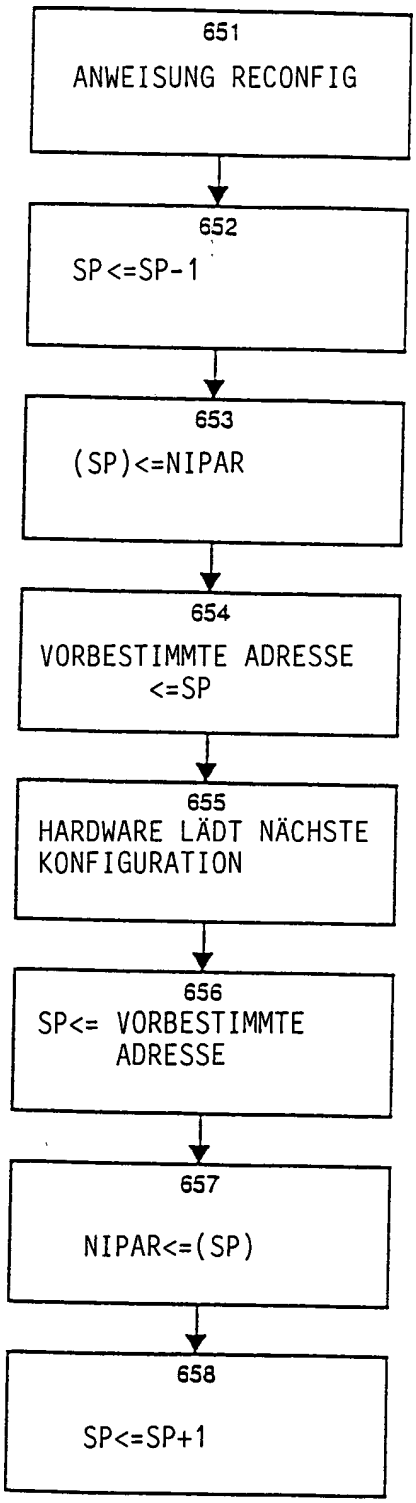
FIGUR 3C


401
QUELLE
401
QUELLE
401
QUELLE
402
KOMPILER
409
ASSEMBLER
403
OBJEKT
403
OBJEKT
403
OBJEKT
404
BINDER
STATISCHE VERBINDUNG
405
AUSFÜHRBARES PROGRAMM
406
ISA-BITSTRÖME
408
DYNAMISCHE VERBINDUNG
10
REKONFIGURIERBARER COMPUTER
407
LADEPROGRAMM
400

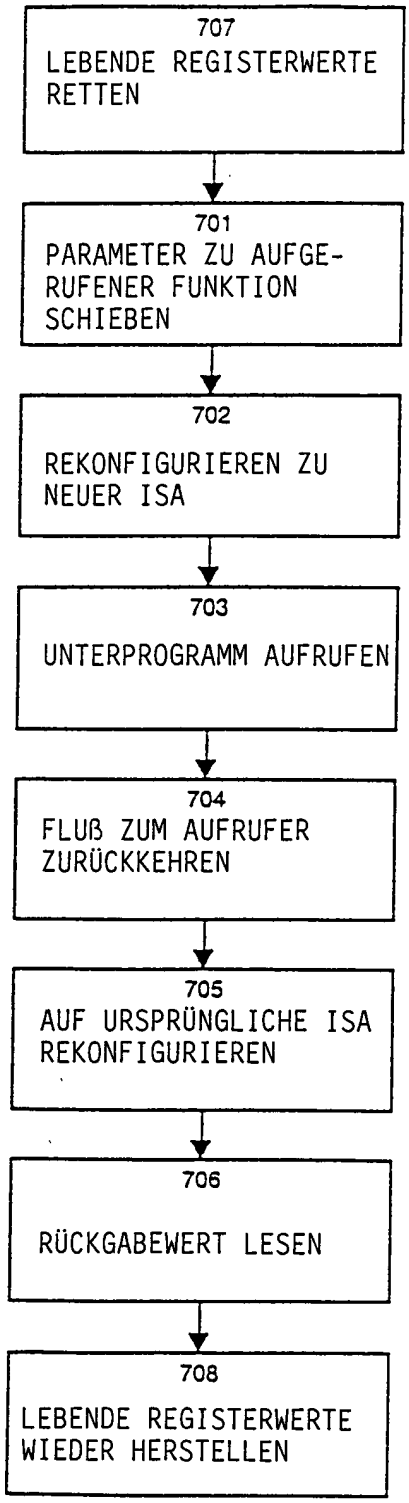
FIGUR 4



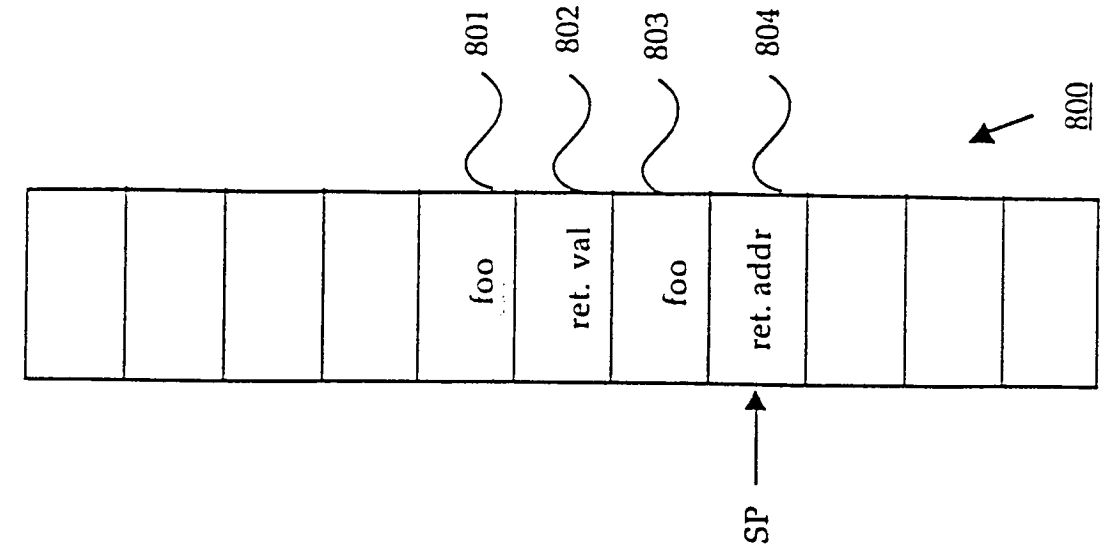
FIGUR 5 (STAND DER TECHNIK)


651
ANWEISUNG RECONFIG
652
SP<=SP-1
653
(SP)<=NIPAR
654
VORBESTIMMTE ADRESSE
<=SP
655
HARDWARE LÄDT NÄCHSTE
KONFIGURATION
656
SP<= VORBESTIMMTE
ADRESSE
657
NIPAR<=(SP)
658
SP<=SP+1

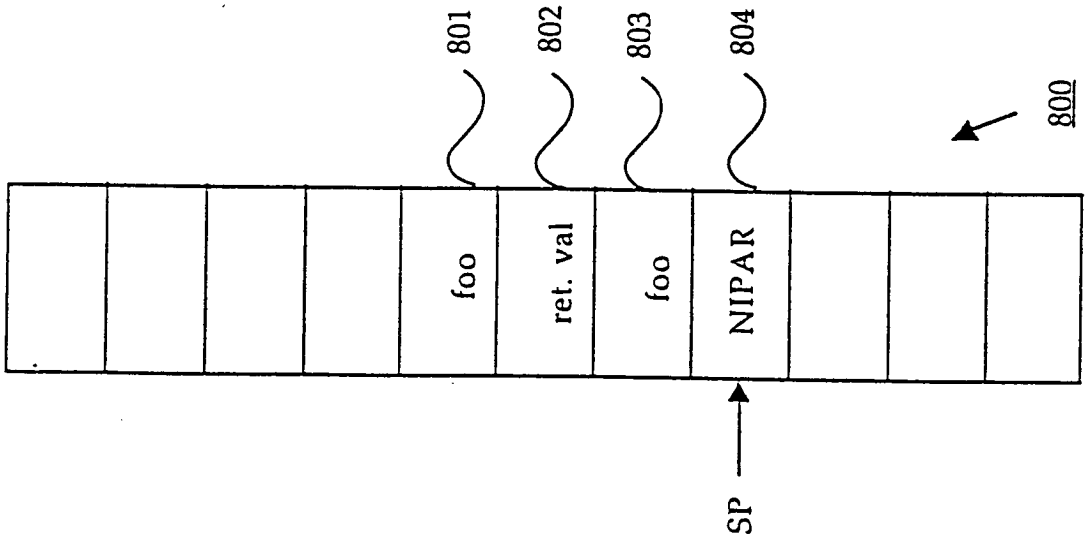
FIGUR 6


707
LEBENDE REGISTERWERTE RETTEN
701
PARAMETER ZU AUFGE-RUFENER FUNKTION SCHIEBEN
702
REKONFIGURIEREN ZU NEUER ISA
703
UNTERPROGRAMM AUFRUFEN
704
FLUß ZUM AUFRUFER ZURÜCKKEHREN
705
AUF URSPRÜNGLICHE ISA REKONFIGURIEREN
706
RÜCKGABEWERT LESEN
708
LEBENDE REGISTERWERTE WIEDER HERSTELLEN

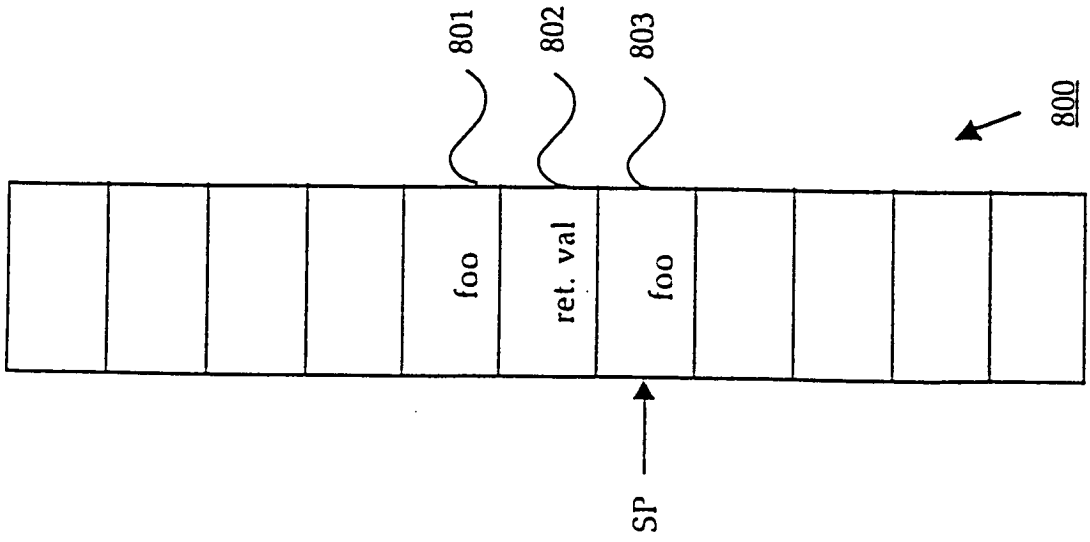
FIGUR 7



FIGUR 8C



FIGUR 8B



FIGUR 8A

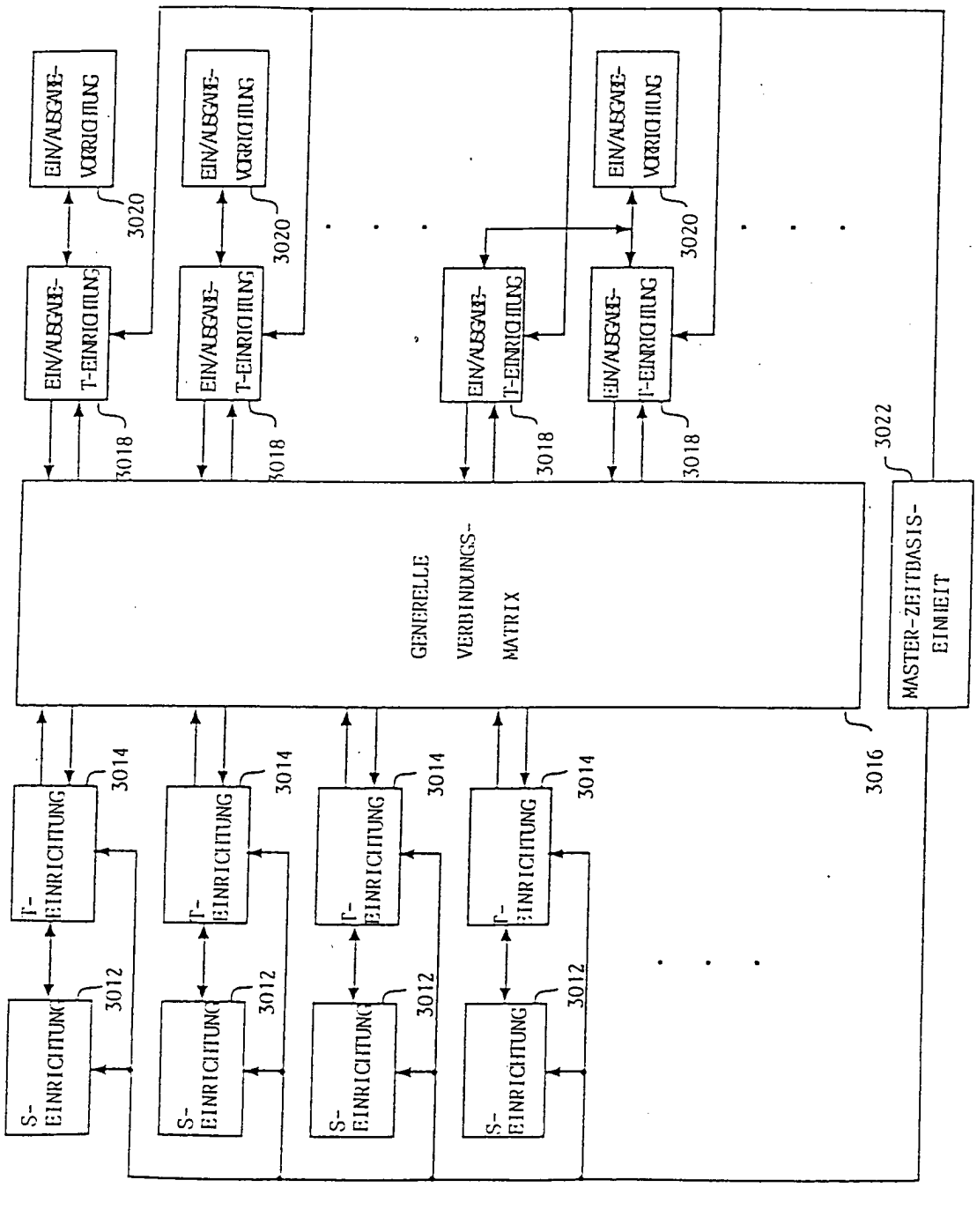

S-EINRICHTUNG
T-EINRICHTUNG
EIN/AUSGABE-VORRICHTUNG
GENERELLE VERBINDUNGS-MATRIX
MASTER-ZEITBASIS-EINHEIT
3010
3012
3014
3016
3018
3020
3022

FIG. 9

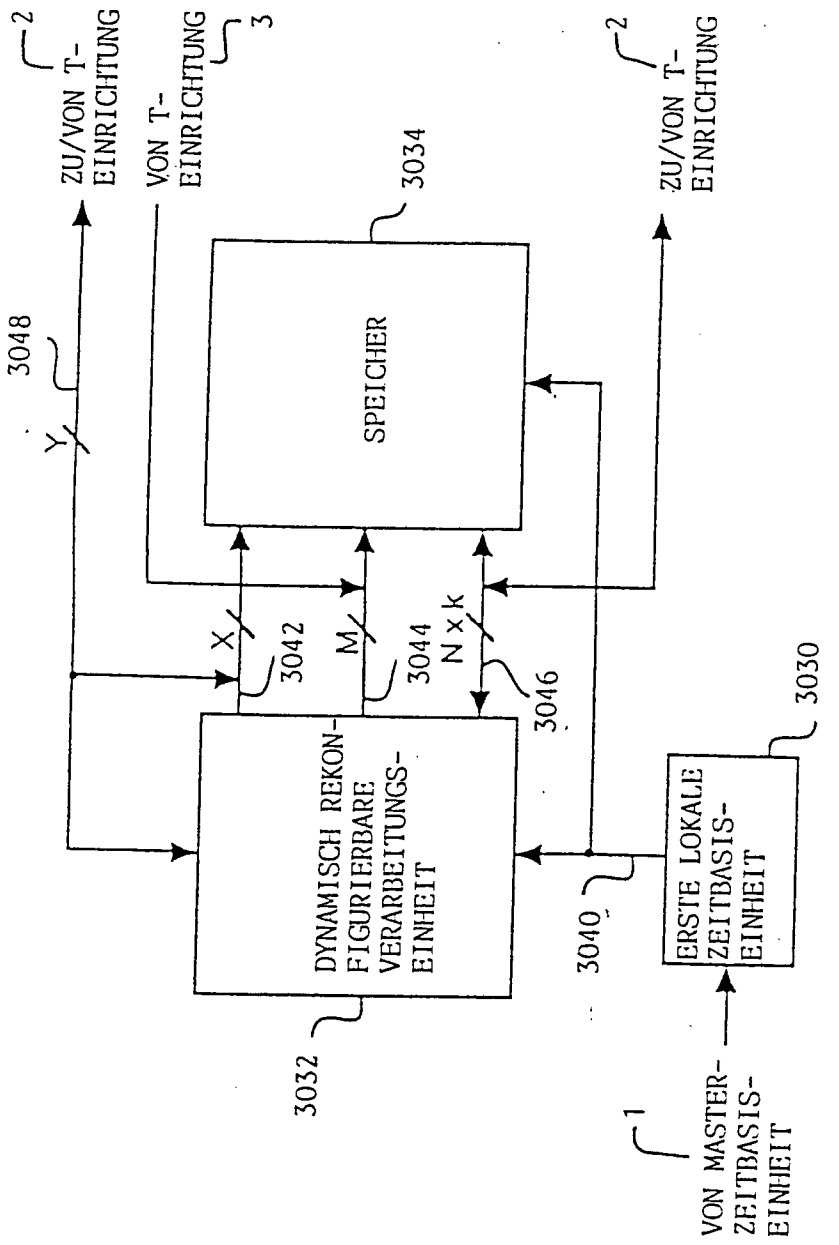

2
ZU/VON T-EINRICHTUNG
VON T-EINRICHTUNG
3
3034
SPEICHER
2
ZU/VON T-EINRICHTUNG
3048
Y
X
3042
M
3044
N x k
3046
DYNAMISCH REKON-FIGURIERBARE VERARBEITUNGS-EINHEIT
3030
ERSTE LOKALE ZEITBASIS-EINHEIT
3040
3032
1
VON MASTER-ZEITBASIS-EINHEIT
3012

FIG. 10

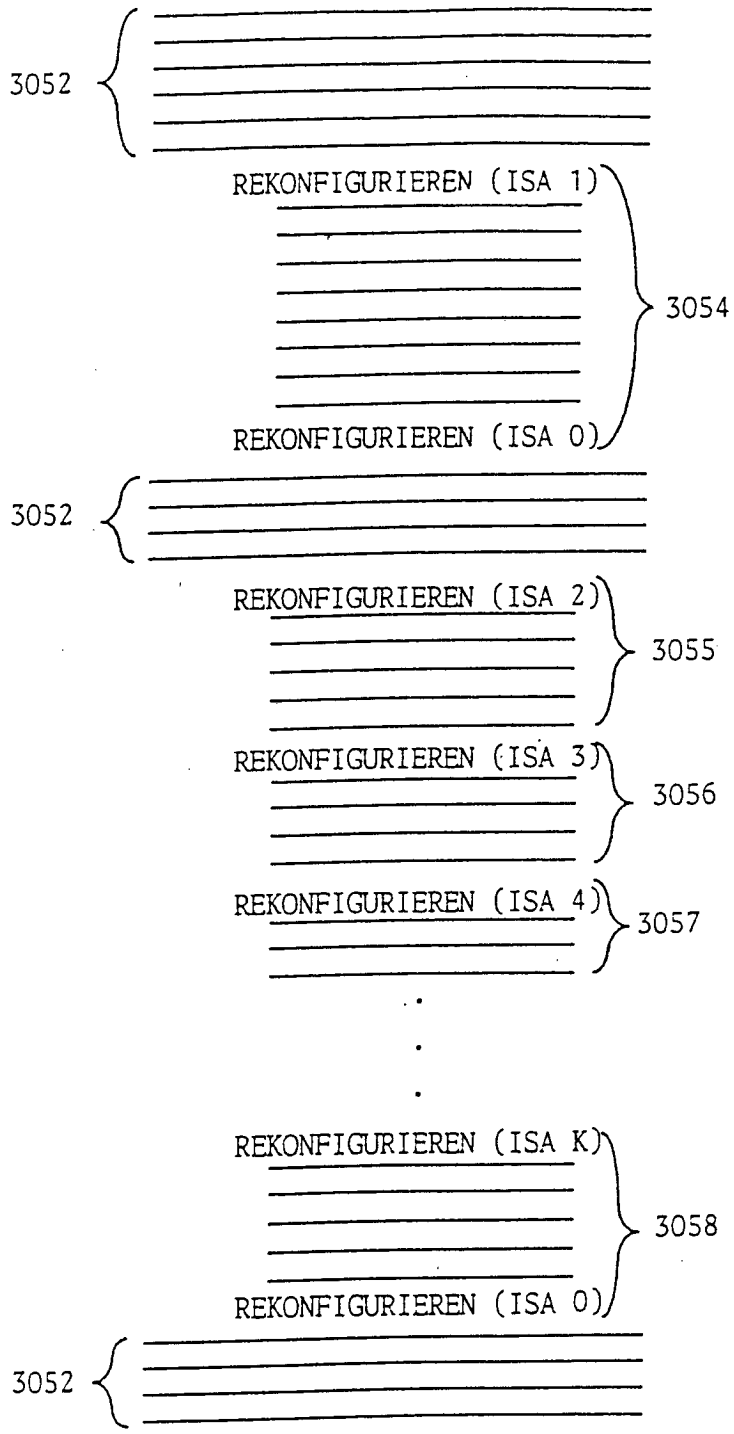

3052
REKONFIGURIEREN (ISA 1)
3054
REKONFIGURIEREN (ISA 0)
3052
REKONFIGURIEREN (ISA 2)
3055
REKONFIGURIEREN (ISA 3)
3056
REKONFIGURIEREN (ISA 4)
3057
REKONFIGURIEREN (ISA K)
3058
REKONFIGURIEREN (ISA 0)
3052
3050

FIG. 11 A

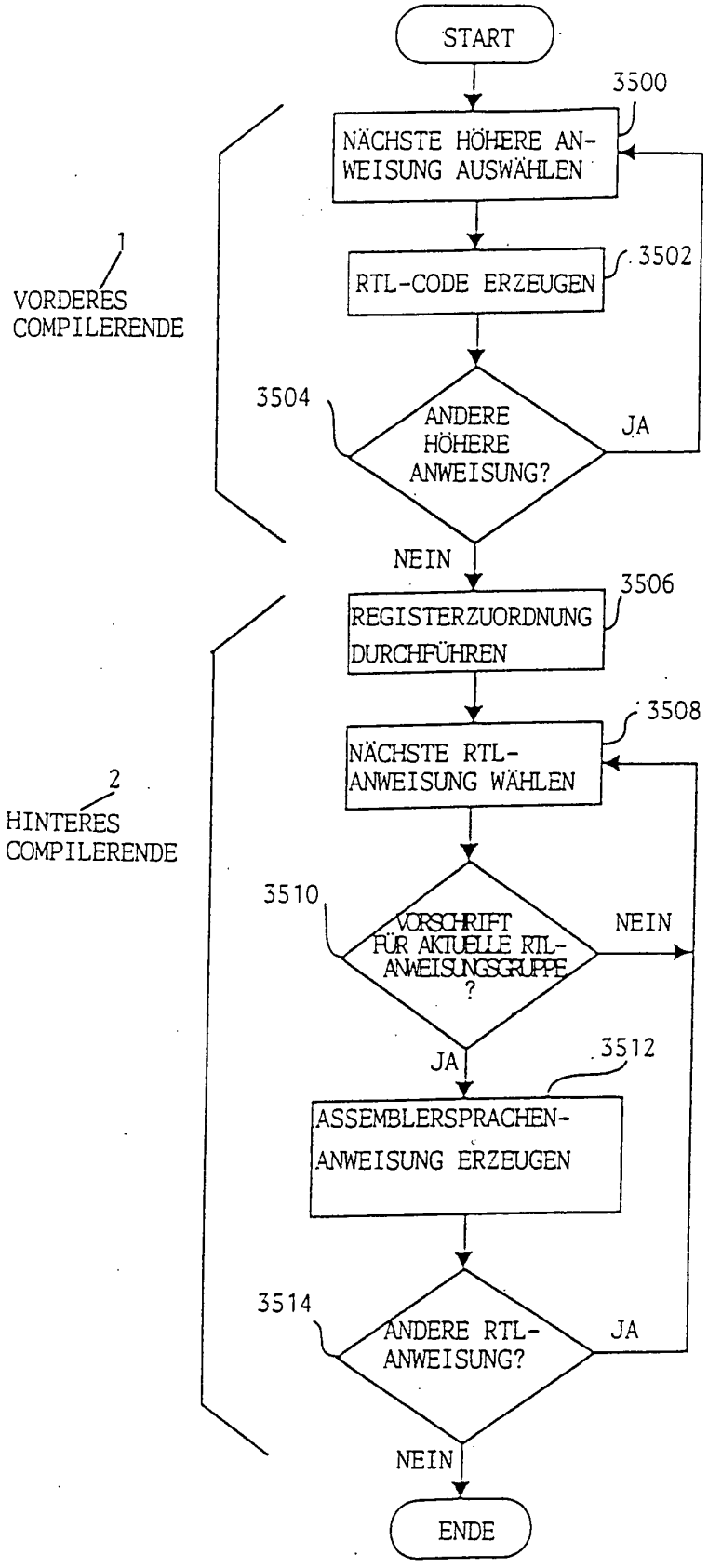

START
3500
NÄCHSTE HÖHERE ANWEISUNG AUSWÄHLEN
1
VORDERES COMPILERENDE
RTL-CODE ERZEUGEN
3502
3504
ANDERE HÖHERE ANWEISUNG?
JA
NEIN
3506
REGISTERZUORDNUNG DURCHFÜHREN
3508
NÄCHSTE RTL-ANWEISUNG WÄHLEN
2
HINTERES COMPILERENDE
3510
VORSCHRIFT FÜR AKTUELLE RTL-ANWEISUNGSGRUPPE ?
NEIN
JA
3512
ASSEMBLERSPRACHEN-ANWEISUNG ERZEUGEN
3514
ANDERE RTL-ANWEISUNG?
JA
NEIN
ENDE

FIG. 11 B

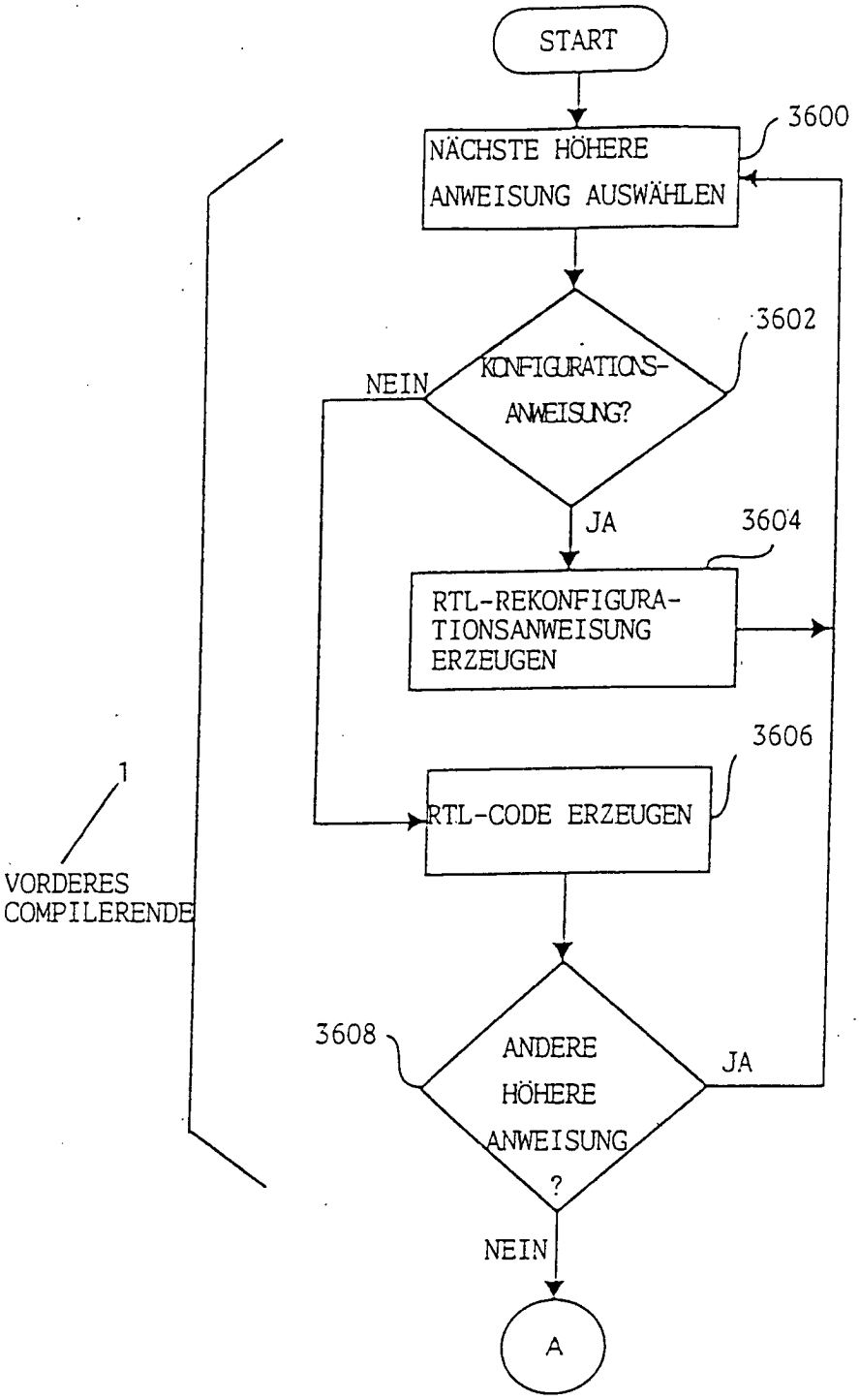

START
NÄCHSTE HÖHERE ANWEISUNG AUSWÄHLEN
3600
KONFIGURATIONS-ANWEISUNG?
3602
NEIN
JA
RTL-REKONFIGURATIONSANWEISUNG ERZEUGEN
3604
RTL-CODE ERZEUGEN
3606
ANDERE HÖHERE ANWEISUNG ?
3608
JA
NEIN
A
1
VORDERES COMPILERENDE

FIG. 11C

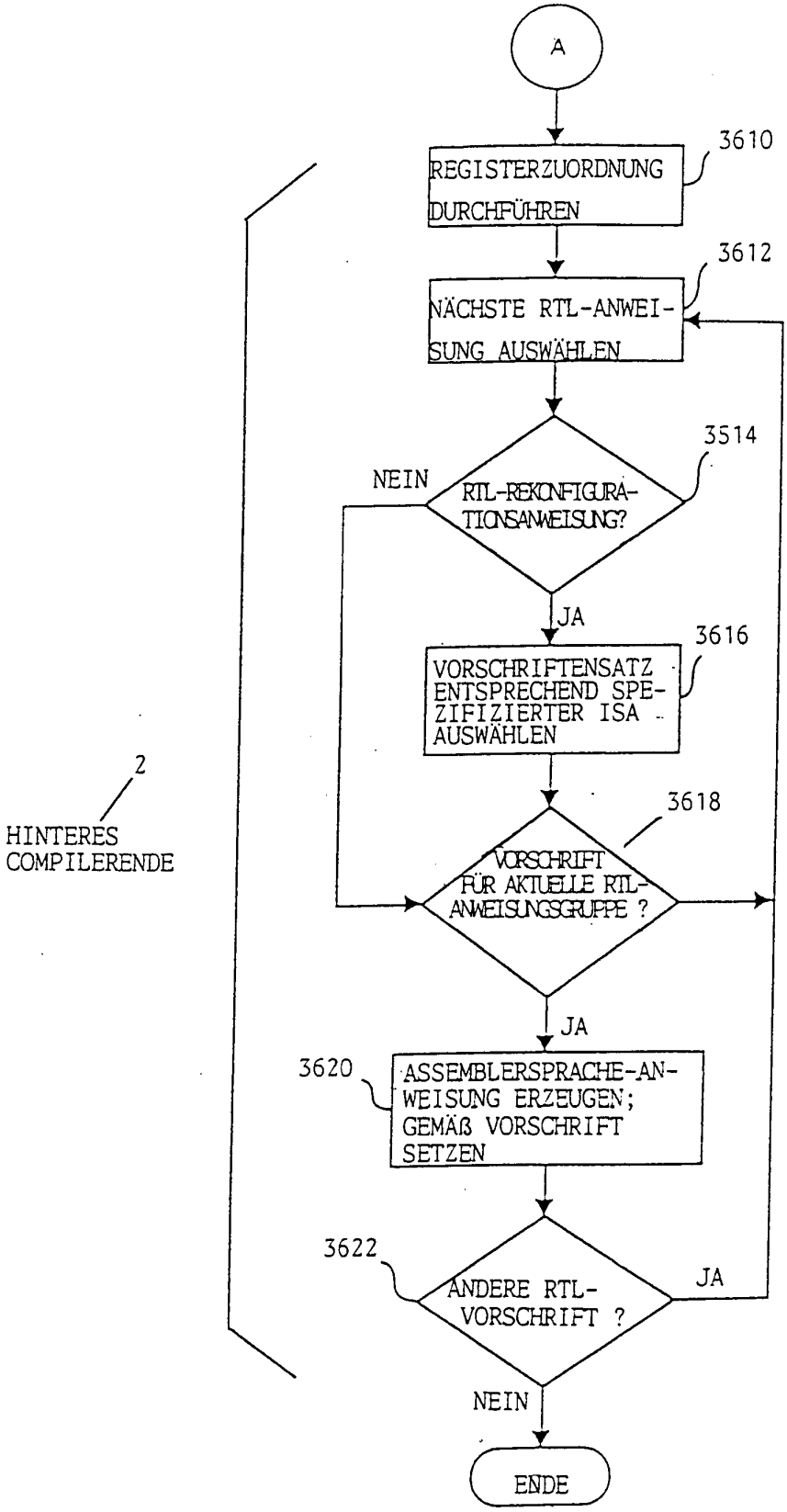

A
3610
REGISTERZUORDNUNG DURCHFÜHREN
3612
NÄCHSTE RTL-ANWEISUNG AUSWÄHLEN
3514
NEIN
RTL-REKONFIGURATIONSANWEISUNG?
JA
3616
VORSCHRIFTENSATZ ENTSPRECHEND SPEZIFIZIERTER ISA AUSWÄHLEN
3618
VORSCHRIFT FÜR AKTUELLE RTL-ANWEISUNGSGRUPPE ?
JA
3620
ASSEMBLERSPRACHE-ANWEISUNG ERZEUGEN; GEMÄß VORSCHRIFT SETZEN
3622
ANDERE RTL-VORSCHRIFT ?
JA
NEIN
ENDE
2
HINTERES COMPILERENDE

FIG. 11 D