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

(10) **DE 698 19 046 T2** 2004.07.22

(12) # Übersetzung der europäischen Patentschrift

(97) **EP 1 187 042 B1**
(21) Deutsches Aktenzeichen: **698 19 046.7**
(96) Europäisches Aktenzeichen: **01 120 244.7**
(96) Europäischer Anmeldetag: **07.08.1998**
(97) Erstveröffentlichung durch das EPA: **13.03.2002**
(97) Veröffentlichungstag
der Patenterteilung beim EPA: **15.10.2003**
(47) Veröffentlichungstag im Patentblatt: **22.07.2004**

(51) Int Cl.[7]: **G06F 17/50**

(30) Unionspriorität:
919531 **28.08.1997** **US**

(84) Benannte Vertragsstaaten:
DE, FR, GB

(73) Patentinhaber:
Xilinx, Inc., San Jose, Calif., US

(72) Erfinder:
Guccione, A, Steven, Austin, US

(74) Vertreter:
Wilhelms, Kilian & Partner, 81541 München

(54) Bezeichnung: **Verfahren für Entwurf von FPGAs für dynamisch rekonfigurierbares Rechnen**

Beschreibung

HINTERGRUND DER ERFINDUNG

Bereich der Erfindung

[0001] Die vorliegende Erfindung betrifft allgemein den Bereich frei programmierbarer logischer Anordnungen (FPGAs) und insbesondere ein Verfahren zum Konfigurieren einer FPGA mit einem Host-Prozessor und einem Compiler einer höheren Programmiersprache.

Beschreibung des technischen Hintergrundes

[0002] In den letzten Jahren bestand zunehmendes Interesse an umkonfigurierbarer logikgestützter Verarbeitung. Diese Systeme arbeiten mit dynamisch umkonfigurierbarer Logik wie z. B. FPGAs, die während des Gebrauchs umkonfiguriert werden kann, um 1 Algorithmen direkt hardwaremäßig zu implementieren, wodurch die Leistung erhöht wird.

[0003] Laut einer Zählung wurden wenigstens 50 verschiedene Hardware-Plattformen (z. B. Computer) zum Untersuchen dieses neuartigen Rechenansatzes gebaut. Leider ist hier die Software hinter der Hardware zurückgeblieben. Die meisten Systeme arbeiten heute mit traditionellen Schaltungsentwurfstechniken, und diese Schaltungen werden dann mittels standardmäßiger Programmiersprachen mit einem Host-Computer verbunden.

[0004] Im Bereich High-Level-Sprachsupport durchgeführte Arbeiten für logikgestütztes umkonfigurierbares Rechnen fallen derzeit in zwei Hauptansatzkategorien. Der erste Ansatz ist, eine traditionelle Programmiersprache anstatt einer Hardware-Beschreibungssprache zu verwenden. Dieser Ansatz erfordert immer noch Software-Support auf dem Host-Prozessor. Der zweite Hauptansatz besteht in der Kompilation standardmäßiger Programmiersprachen zu umkonfigurierbaren Logik-Koprozessoren. Bei diesen Ansätzen wird gewöhnlich versucht, rechenintensive Code-Teile zu erfassen und sie auf den Koprozessor abzubilden. Diese Kompilationstools sind jedoch gewöhnlich an traditionelles Platzieren und Routen von Backends gebunden und haben relativ lange Kompilationszeiten. So offenbart z. B. das Dokument WO-A-94 10627 (GIGA OPERATIONS CORP; TAYLOR BRAD (US); DOWLING ROBERT (US), veröffentlicht am 11. Mai 1994, ein System programmierbarer Logikbauelemente (PLDs) zum Implementieren eines Programms in Hardware, wobei der Host-C-Compiler Host-C-Sourcecode in ablauffähigen binären Host-Code kompiliert, während der separate PLD-C-Sourcecode, der die Platzierungs- und Routinginformationen enthält, mit dem PLD-C-Compiler zu Konfigurationsdaten kompiliert werden muss, die auf einem PLD laufen können. Sie bieten auch wenig oder gar keinen Laufzeit-Support für eine dynamische Umkonfiguration.

[0005] Im Allgemeinen basieren heutige Tools auf statischen Schaltungsdesigntools, die ursprünglich für den Einsatz beim Entwerfen von Leiterplatten und integrierten Schaltungen entwickelt wurden. Das volle Potential von dynamischer Logik wird von solchen statischen Design-Tools nicht unterstützt.

ZUSAMMENFASSUNG DER ERFINDUNG

[0006] Gemäß der vorliegenden Erfindung wird ein Verfahren zum Konfigurieren einer frei programmierbaren logischen Anordnung (FPGA) (**106**) zum dynamisch umkonfigurierbaren Rechnen bereitgestellt, wobei das Verfahren die folgenden Schritte umfasst:

a) Programmieren des Host-Prozessors mit Anweisungen (**201**) in einer höheren Programmiersprache;

b) Bereitstellen eines Compilers (**203**) für die höhere Programmiersprache, der auf dem Host-Computer läuft, um ablauffähigen Code (**204**) als Reaktion auf die Anweisungen (**201**) zu erzeugen, wobei der ablauffähige Code (**204**) kompilierte Platzierungs- und Leitweginformationen beinhaltet; und

c) Verbinden des Host-Prozessors mit der FPGA (**106**) für eine dynamische Umkonfigurationsprogrammierung der FPGA (**106**) durch den Host-Prozessor über den ablauffähigen Code (**204**).

[0007] Die höhere Programmiersprache kann Java sein. Die Erfindung kann den weiteren Schritt des Instanziierens von Elementen von einer Bibliothek von Elementen umfassen, die mit dem Compiler kompatibel sind. Die Bibliothek kann kombinatorische Logikelemente oder Flipflops oder Latch-Elemente umfassen.

[0008] Das erfindungsgemäße Entwurfsverfahren für umkonfigurierbares Rechnen (MDRC) bedeutet einen neuartigen Ansatz für ein Hardware/Software-Co-Design für umkonfigurierbare logikgestützte Koprozessoren. Es können ein System und ein Verfahren zum Konfigurieren einer FPGA direkt von einem Host-Prozessor bereitgestellt werden. Es ist nicht notwendig, die Konfigurationsdaten in einer Datei zu speichern, obwohl sie bei Bedarf dort gespeichert werden können.

[0009] Daher ist dieses Verfahren besonders für den Einsatz mit FPGAs wie umkonfigurierbare Koprozessoren geeignet, die häufig "im Vorbeigehen" umkonfiguriert werden, d. h. ohne die FPGA zu überwältigen, und zuweilen während nur ein Teil der FPGA umkonfiguriert wird. Die gewünschte Funktionalität für die FPGA wird unter Verwendung der MDRC-Bibliotheken und einer standardmäßigen höheren Progranmiersprache wie Java™ (Java ist ein Warenzeichnen von Sun Microsystems, Inc.) beschrieben. Konfiguration, Umkonfiguration und Host-Schnittstellensoftware für umkonfigurierbare Koprozessoren werden in einem einzigen Code-Teil

unterstützt.

[0010] Da MDRC nicht von dem traditionellem Platzierungs- und Leitwegansatz der Schaltungssynthese Gebrauch macht, sind die Kompilationszeiten erheblich kürzer als bei Verfahren des Standes der Technik und liegen in der Größenordnung von Sekunden. Diese schnelle Kompilation ergibt eine Entwicklungsumgebung, die der für eine moderne Software-Entwicklung verwendeten ähnlich ist.

[0011] MDRC bietet eine einfache Alternative zu einem Design auf der Basis eines traditionellen Computer Aided Design (CAD) Tools. In der bevorzugten Ausgestaltung werden Java-Bibliotheken zum Programmieren eines FPGA-Bauelementes verwendet. Dieses Verfahren hat die folgenden Vorteile:

[0012] Sehr schnelle Kompilationszeiten. Da bei diesem Ansatz Compiler mit standardmäßiger Programmiersprache verwendet werden, ist die Kompilation so schnell wie beim systemorientierten Host-Compiler. Mit derzeitigen Java-Compilern wie dem J++ 1.1 Compiler von Microsoft, der über 10.000 Zeilen Code pro Sekunde kompiliert, erfordert die Kompilation von Schaltungen, die unter Verwendung von MDRC erstellt werden, insgesamt etwa eine Sekunde. Dies steht im Gegensatz zu einem Zeitaufwand von Stunden bei existierenden CAD-Tools.

[0013] Laufzeitparameterisierung von Schaltungen. Vielleicht das interessanteste Merkmal von MDRC ist seine Fähigkeit, Laufzeitparameterisierung von Schaltungen durchzuführen. So kann z. B. ein Konstantenaddierer unter Verwendung eines nur zur Laufzeit bekannten konstanten Wertes während der Ausführung von MDRC konfiguriert werden. Die Größe einer bestimmten Komponente kann ebenfalls dynamisch vorgegeben werden. Ein 5-Bit-Addierer oder ein 9-Bit-Zähler kann beispielsweise zur Laufzeit konfiguriert werden. Dieses Merkmal wird in Bereichen wie adaptive Filterung eingesetzt.

[0014] Objektorientiertes Hardware-Design. Da Java eine objektorientierte Sprache ist (d. h. eine strukturierte Sprache, in der Elemente im Sinne von Objekten und den Verbindungen zwischen diesen Objekten beschrieben werden), kann in dieser Sprache konstruierte Hardware objektorientierten Support nutzen. Mit MDRC konstruierte Bibliotheken können als Objekte verpackt und wie jede standardmäßige Software-Komponente manipuliert und wiederverwendet werden.

[0015] Unterstützung für dynamische Umkonfiguration. Die Fähigkeit, eine Schaltung automatisch dynamisch zu konfigurieren, bringt die Möglichkeit einer dynamischen Umkonfiguration mit sich. Verwendungszwecke für diese Fähigkeit sind bereits im Kommen. So könnte man z. B. einen Teil einer dynamisch umkonfigurierbaren FPGA als Vervielfacher konfigurieren, der einen Eingangswert mit einer Konstante vervielfacht, wobei die Konstante ein Skalierungsfaktor in einer Signalverarbeitungsanwendung ist. Mit dynamischer Umkonfiguration kann dieser Skalierungsfaktor verändert werden, ohne die Funktion anderer Teile der konfigurierten FPGA zu unterbrechen.

[0016] Standardmäßige Software-Entwicklungsumgebung. Mit einer standardmäßigen Programmiersprache (in diesem Fall Java) können Schaltungsentwickler standardmäßige Software-Umgebungen verwenden. Mit anderen Worten, im Handel leicht erhältliche Compiler wie der J++ 1.1 Compiler von Microsoft könnten für die Entwicklung von Schaltungen verwendet werden, die in einer FPGA implementiert werden sollen. Diese Fähigkeit hat zwei unmittelbare Vorteile. Erstens, der Benutzer kann das Tool weiter verwenden, mit dem er/sie bereits vertraut ist. Zweitens, und das ist möglicherweise am wichtigsten, FPGA-Design wird zu einem Software-Entwicklungsbemühen, das für Programmierer offen ist. Diese Fähigkeit könnte die existierende Basis von FPGA-Benutzern stark erweitern.

[0017] Vereinfachtes Host-Interfacing. MDRC verlangt, dass ein Host-Prozessor für die Ausführung des Java-Codes und die Lieferung von Konfigurationsdaten zur FPGA zur Verfügung steht. Diese Kombination aus Prozessor und FPGA ist eine leistungsstarke Coprocessing-Umgebung, die derzeit von Forschern untersucht wird. Ein Hindernis für die Verwendung dieser Systeme ist die Notwendigkeit, das FPGA-Hardware-Design mit dem Host-Software-Design zu verbinden. MDRC führt Software- und Hardware-Design-Aktivitäten zu einer einzigen Aktivität zusammen, so dass diese Schnittstellenprobleme wegfallen.

[0018] Flexibilität. Da MDRC eine Bibliothek umfasst, die von einer standardmäßigen Programmiersprache verwendet wird, kann sie – sogar von Benutzern – erweitert werden. Diese Fähigkeit bietet ein Niveau an Flexibilität, das es in statischen Design-Tools bisher nicht gab. Benutzer können neue Bibliotheken und Bibliothekselemente oder sogar Zubehör wie kundenspezifische grafische Benutzeroberflächen bereitstellen.

[0019] Standardmäßige Bauelementschnittstelle. Eine Art, sich MDRC vorzustellen, ist nicht so sehr als Tool an sich, sondern als Standardschnittstelle zum FPGA-Bauelement. Diese Schnittstelle kann für die FPGA-Konfiguration oder auch zum Schaffen anderer Tools verwendet werden. MDRC kann sogar als Basis für traditionelle CAD-Software wie Platzierungs- und Leitwegtools benutzt werden. Eine andere Art, sich MDRC vorzustellen, ist als "Assembler-Sprache" der FPGA.

[0020] Garantiert "sichere" Schaltungen. MDRC bietet eine Abstraktion (ein Software-Konstrukt, das Hardware – häufig vereinfacht – darstellt), mit der keine Schaltungen mit Konkurrenzproblemen erzeugt werden können. So wird es bei Verwendung von MDRC unmöglich, dass das Bauelement aufgrund einer schlechten Konfiguration aus Versehen beschädigt oder zerstört wird. Ein solcher Schutz ist in einer dynamischen Programmierumgebung wie MDRC äußerst wünschenswert, wo ein Programmierfehler

sonst zu dauerhaften Hardwareschäden führen könnte. (Eine falsch konfigurierte FPGA kann zu einem versehentlichen Kurzschluss von Leistung und Masse führen, wodurch das Bauelement zerstört wird.) Ein Nebeneffekt dieses Merkmals ist, dass MDRC als Implementationsvehikel für das in der Entstehung befindliche Feld genetischer Algorithmen verwendet werden kann.

KURZE BESCHREIBUNG DER ZEICHNUNGEN

[0021] Die oben genannten Aufgaben und Vorteile sowie weitere Aufgaben und Vorteile der vorliegenden Erfindung werden nach einem Studium der folgenden ausführlichen Beschreibung einer bevorzugten Ausgestaltung in Verbindung mit den nachfolgenden Zeichnungen besser verständlich.

[0022] **Fig.** 1 ist ein Blockdiagramm, das den Designablauf des Standes der Technik zum Konstruieren einer Schaltung illustriert, die in einer FPGA mit einem umkonfigurierbaren Logik-Koprozessor implementiert wird.

[0023] **Fig.** 2 ist ein Blockdiagramm, das den Designablauf in der vorliegenden Erfindung illustriert.

[0024] **Fig.** 3 ist ein Diagramm einer Logikzellenabstraktion der vorliegenden Erfindung auf Level **1**.

[0025] **Fig.** 3A ist ein Diagramm einer XC6200 Logikzelle, die durch die Abstraktion von **Fig.** 3 repräsentiert wird.

[0026] **Fig.** 4 ist ein Diagramm eines Mehrbit-Zählers gemäß einer Ausgestaltung der Erfindung.

[0027] **Fig.** 5 ist eine Elementdefinitionscodeliste für die Grundelemente der Ausgestaltung von **Fig.** 4.

[0028] **Fig.** 6A ist ein Diagramm einer Toggle-Flipflopzelle der Ausgestaltung von **Fig.** 4.

[0029] **Fig.** 6B ist ein Diagramm einer Übertragslogikzelle der Ausgestaltung von **Fig.** 4.

[0030] **Fig.** 7 ist eine Konfigurationscodeliste für den Zähler von **Fig.** 4.

[0031] **Fig.** 8A ist Laufzeitcode für den Zähler von **Fig.** 4.

[0032] **Fig.** 8B ist eine Ausführungskurve für den Zähler von **Fig.** 4.

AUSFÜHRLICHE BESCHREIBUNG DER ZEICHNUNGEN

[0033] Das Entwerfen einer in einer FPGA implementierten Schaltung mit einem umkonfigurierbaren Logik-Koprozessor erfordert derzeit eine Kombination von zwei verschiedenen Entwurfswegen, wie in **Fig.** 1 (Stand der Technik) dargestellt ist. Der erste und möglicherweise signifikanteste Teil der Arbeit beinhaltet das Schaltungsdesign mit traditionellen CAD-Tools. Der Entwurfweg für diese CAD-Tools beinhaltet gewöhnlich die Eingabe eines Designs **101** mit einem Schema-Editor oder einer Hardware-Beschreibungssprache (HDL), die Verwendung eines Netlisters **102** zum Erzeugen einer Netzliste **103** für das Design, das Importieren dieser Netzliste in ein FPGA-Platzierungs- und -Routingtool **104**, das schließlich eine Bitstream-Datei **105** von Konfigurationsdaten erzeugt, die zum Konfigurieren der FPGA **106** verwendet wird.

[0034] Nach dem Erzeugen der Konfigurationsdaten besteht die nächste Aufgabe darin, Software bereitzustellen, die den Host-Prozessor mit der FPGA verbindet. Der Benutzer gibt den Benutzercode **107** ein, der die Benutzeroberflächenanweisungen beschreibt und der dann mit dem Compiler **108** zum Erzeugen von ablauffähigem Code **109** kompiliert wird. Die Anweisungen in ablauffähigem Code **109** werden dann vom Prozessor für die Kommunikation mit der konfigurierten FPGA **106** verwendet. Es ist auch bekannt, ablauffähigen Code **109** zu verwenden, um die Konfiguration von FPGA **106** mit der Bitstream-Datei **105** zu steuern. Diese Serie von Aufgaben wird gewöhnlich vollständig von der Aufgabe des Entwerfens der Schaltung abgekoppelt und kann somit schwierig und fehleranfällig sein.

[0035] Zusätzlich zu den Problemen in Verbindung mit dem Interfacing von Hardware und Software in dieser Umgebung besteht auch das Problem der Design-Zykluszeit. Jede Änderung des Schaltungsentwurfs erfordert einen kompletten Durchgang durch die Hardware-Designtool-Kette (**101**–**106** in **Fig.** 1). Dieser Vorgang ist zeitaufwändig, und der Platzierungs- und Routingteil der Kette braucht gewöhnlich mehrere Stunden.

[0036] Schließlich unterstützt dieser Ansatz keine Umkonfiguration. Die traditionellen Hardware-Designtools bieten Unterstützung fast ausschließlich für statisches Design. Man kann sich nur schwer Konstrukte vorzustellen, die Laufzeitumkonfiguration in Umgebungen auf der Basis einer schematischen oder HDL-Design-Eingabe unterstützen.

[0037] Im Gegensatz dazu umfasst die MDRC-Umgebung eine Bibliothek von Elementen, mit denen Logik und Routing in einem umkonfigurierbaren Logikbauelement vorgegeben und konfiguriert werden können. Mittels Aufrufen an diese Bibliothekselemente können Schaltungen konfiguriert und umkonfiguriert werden. Ferner kann Host-Code so geschrieben werden, dass er mit der umkonfigurierbaren Hardware interagiert. Dadurch können sich alle Designdaten in einem einzelnen System befinden, häufig in einer einzigen Java-Quellcode-Datei.

[0038] Der MDRC-Ansatz vereinfacht nicht nur den Designablauf sehr stark, wie in **Fig.** 2 gezeigt ist, sondern verbindet auch die Hardware- und Software-Entwurfsprozesse eng miteinander. Design-Parameter für die umkonfigurierbare Hardware und die Host-Software werden gemeinsam genutzt. Diese Kopplung bietet eine bessere Unterstützung für die Aufgabe des Verbindens der Logikschaltungen mit der Software.

[0039] Wie in **Fig.** 2 gezeigt, erfordert das Eingeben und Kompilieren einer FPGA-Schaltung mit dem MDRC-Verfahren viel weniger Schritte als im Verfahren des Standes der Technik gemäß **Fig.** 1. Benut-

zercode **201**, in dieser Ausgestaltung Java-Code, wird eingegeben. Dieser Code beinhaltet nicht nur Anweisungen, die die Benutzeroberfläche und den Konfigurationsprozess beschreiben, sondern auch eine High-Level-Beschreibung der gewünschten FPGA-Schaltung. Diese Schaltungsbeschreibung umfasst Aufrufe an Bibliothekselemente (Funktionsaufrufe) in MDRC-Bibliotheken **202**. In einer Ausgestaltung können diese Zellen parameterisiert werden. Der Java-Compiler **203** kombiniert die Schaltungsbeschreibungen von MDRC-Bibliotheken **202** mit den Anweisungen vom Benutzercode **201** zum Erzeugen von ablauffähigem Code **204**. Ablauffähiger Code **204** beinhaltet nicht nur Benutzeroberflächenanweisungen, wie im ablauffähige Code **109** von **Fig.** 1, sondern Konfigurationsanweisungen. Bei Verwendung von MDRC braucht der Bitstream nicht als Datei gespeichert zu werden; falls gewünscht, können die Konfigurationsdaten direkt von ablauffähigem Code **204** auf die FPGA **106** heruntergeladen werden. Diese Technik ist besonders beim umkonfigurierbaren Rechnen nützlich, d. h. bei Verwendung einer umkonfigurierbaren FPGA als Koprozessor zum Durchführen einer Reihe verschiedener Rechnungen für einen Mikroprozessor.

Die MDRC-Abstraktion

[0040] MDRC repräsentiert die umkonfigurierbare Logik in einem mehrschichtigen Ansatz. In der untersten (ausführlichsten) Schicht, Level 0 genannt, unterstützt MDRC alle zugängigen Hardware-Ressourcen in der umkonfigurierbaren Logik. Eine umfangreiche Verwendung von Konstanten und anderen symbolischen Daten macht Level 0 trotz der notwendigerweise niedrigen Abstraktionsebene benutzbar.

[0041] Die derzeitige Plattform für die MDRC-Umgebung ist das von Xilinx, Inc., der Zessionarin der vorliegenden Erfindung, hergestellte Entwicklungssystem XC6200DS. Das Entwicklungssystem XC6200DS umfasst eine PCI-Platte mit einer Xilinx XC6216 FPGA. In der FPGA-Familie XC6200 umfasst Level 0 Support Abstraktionen für die umkonfigurierbaren Logikzellen und alle Routing-Schalter einschließlich Taktrouting. Der Code für Level 0 ist im Wesentlichen die Bit-Level-Information in dem in Java codierten XC6200 Data Sheet (das "XC6200 Data Sheet", auf das hierin verwiesen wird, umfasst Seiten 4–251 bis 4–286 des Xilinx 1996 Data Book mit dem Titel "The Programmable Logic Data Book", herausgegeben im September 1996, erhältlich von Xilinx, Inc., 2100 Logic Drive, San Jose, Kalifornien 95124 (Xilinx Inc., Inhaber der Urheberrechte, hat keine Einwände gegen ein Kopieren dieser und anderer Seiten, auf die hierin verwiesen wird, behält sich aber ansonsten alle Urheberrechte vor).

[0042] Level 0 unterstützt zwar im vollen Umfang das Konfigurieren aller Aspekte des Bauelementes, aber sie ist doch sehr tief, ist möglicherweise zu aufwändig und verlangt für die meisten Benutzer zuviel Spezialkenntnis über die Architektur. Diese Schicht steht zwar dem Programmierer immer zur Verfügung, aber es wird erwartet, dass Level 0 Support hauptsächlich als Basis für die höheren Abstraktionsschichten dient. In diesem Sinn ist Level 0 die "Assembler-Sprache" des MDRC-Systems.

[0043] Über Abstraktionslevel 0 liegt Abstraktionslevel 1. Diese Abstraktionsebene vereinfacht den Zugang zu Logikdefinition, Takt- und Lösch-Routing und die Host-Schnittstelle.

[0044] Der signifikanteste Teil von Abstraktionslevel 1 ist die Logikzellendefinition. Mit der Logikzellendefinition kann eine Logikzelle im XC6200-Bauelement zu einem standardmäßigen Logik-Operator konfiguriert werden. In einer Ausgestaltung werden die kombinatorischen Logikelemente AND, NAND, OR, NOR, XOR, XNOR, BUFFER und INVERTER unterstützt. Diese Elemente können einen optionalen registrierten Ausgang haben. Zusätzlich werden ein D-Flipflop und eine Registerlogikzelle definiert. In einer Ausgestaltung wird eine Latch-Zelle anstatt oder zusätzlich zu dem Flipflop-Element definiert. Alle diese Logik-Operatoren werden ausschließlich mit MDRC-Operationen auf Level 0 definiert und lassen sich somit leicht erweitern.

[0045] **Fig.** 3 ist ein Diagramm der Logikzellenabstraktion auf Level 1. Ausgänge Nout, Eout, Sout, Wout entsprechen den gleichnamigen Ausgängen in der XC6200-Logikzelle, wie auf Seite 4–256 des XC6200 Data Sheet abgebildet ist. Die XC6200-Logikzelle ist auch in **Fig.** 3A hierin dargestellt. Eingang Sin von **Fig.** 3 entspricht Eingang S der Logikzelle von **Fig.** 3A, Eingang Win entspricht Eingang W, Nin entspricht N und Ein entspricht E. Das in **Fig.** 3 gezeigte Abstraktionslevel 1 ist eine vereinfachte Darstellung des XC6200-Logikblocks. In dieser Ausgestaltung gibt es z. B. für Eingänge S4, W4, N4 und E4 auf Abstraktionslevel 1 keinen Support, auf Abstraktionslevel **0** aber schon. Der in **Fig.** 3 gezeigte Logikblock und der Flipflop bedeuten die Schaltungen, die in einer XC6200-Logikzelle zur Verfügung stehen. Eingänge A, B und SEL in **Fig.** 3 (entsprechend Eingängen X1, X2 und X3 von **Fig.** 3A) sind die Eingänge zum Logikblock; sie können auf beliebige Logikzelleneingänge Sin, Win, Nin und Ein abgebildet werden. Die in einer Logikzelle verfügbaren Schaltungen sind in anderen FPGA-Bauelementen anders.

[0046] Zusätzlich zur Logikzellenabstraktion wird das Takt-Routing abstrahiert. Verschiedene globale und lokale Taktsignale (wie z. B. Clk und Clr in **Fig.** 3) können definiert und mit einer bestimmten Logikzelle assoziiert werden.

[0047] Ein dritter Teil der MDRC-Abstraktion auf Level 1 ist die Registerschnittstelle. Im XC6200-Bauelement können Spalten von Zellen über die Busschnittstelle gelesen oder geschrieben werden, wobei die Spalten von Zellen somit Lese-/Schreibregister bilden. Die Registerschnittstelle erlaubt eine(n) symbolische(n) Konstruktion der und Zugriff auf die Regis-

ter.

Ein Beispiel

[0048] **Fig.** 4 zeigt einen einfachen Zähler, der für ein XC6200-Bauelement entworfen wurde, auf der Basis von Toggle-Flipflops **402** sowie Übertragslogik **401** mit Abstraktionslevel 1. In weniger als 30 Zeilen Code wird die Schaltung beschrieben und konfiguriert und der Zählerwert wird getaktet und gelesen. Zusätzlich erlaubt die Struktur dieser Schaltung eine einfache Verpackung als parameterisiertes Objekt, wobei die Zahl der Bits in dem Zähler über einen anwenderdefinierten Parameter eingestellt wird. Ein solcher objektgestützter Ansatz würde es zulassen, dass Zähler beliebiger Größe vorgegeben und an einer beliebigen Stelle im XC6200-Bauelement platziert werden. Nach der Implementation könnte auch der Zähler von **Fig.** 4 in einer Bibliothek von parameterisierten Makrozellen platziert werden.

[0049] Der Implementationsprozess ist recht einfach. Zunächst werden die von der Schaltung benötigten Logikelemente definiert. Diese Schaltungselementdefinitionen sind Abstraktionen und sind nicht mit einer bestimmten Hardware-Implementation assoziiert. Nach dem Definieren dieser Logikelemente können sie auf die Hardware geschrieben werden, wodurch die Schaltung konfiguriert wird. Nach dem Konfigurieren der Schaltung erfolgt ein Laufzeit-Interfacing der Schaltung, gewöhnlich in der Form des Lesens und Schreibens von Registern und des Taktens der Schaltung. Wenn es die Anwendung erfordert, kann der Prozess wiederholt werden, wobei die Hardware nach Bedarf umkonfiguriert wird.

[0050] Das Zählerbeispiel enthält neun Grundelemente. Fünf Grundelemente stellen den gesamten notwendigen Unterstützungsschaltkomplex zum Lesen, Schreiben, Takten und Löschen der Hardware bereit. Die übrigen Grundelemente dienen zum Definieren der Zählerschaltung selbst. Diese Elemente sieht man am besten, wenn man **Fig.** 5 in Verbindung mit **Fig.** 4 betrachtet. **Fig.** 5 führt den MDRC-Code zum Beschreiben der Grundelemente auf. Das zu jeder der beiden Registerdefinitionen geleitete pci6200-Objekt ist die Hardware-Schnittstelle zur XC6200DS PCI-Karte.

[0051] Der Unterstützungsschaltkomplex enthält zwei Register, die einfach die Schaltung mit der Host-Software verbinden. Diese beiden Register dienen zum Lesen des Wertes des Zählers ("Register counterReg" in **Fig.** 5) und zum Umschalten eines einzigen Flipflops **404**, so dass der lokale Takt erzeugt wird ("Register clockReg" in **Fig.** 5). Um die Flipflops im XC6200-Bauelement zu unterstützen, müssen auch Takt- und Lösch- (Reset) Eingänge definiert werden. Der globale Takt ("ClockMux globalClock" in **Fig.** 5) ist der Systemtakt für das Bauelement und muss als Takteingang zu einem beschreibbaren Register verwendet werden. In dieser Schaltung muss der Flipflop, der den Software-gesteuerten lokalen Takt erzeugt, den globalen Takt verwenden. Der lokale Takt ("ClockMux localClock" in **Fig.** 5) ist der Ausgang des Software-gesteuerten Taktgebers und muss zu den Toggle-Flipflops geleitet werden, die den Zähler bilden. Schließlich benötigen alle Flipflops in dem XC6200-Bauelement einen Löscheingang ("ClearMux clear" in **Fig.** 5). In dieser Ausgestaltung wird der Löscheingang zu allen Flipflops einfach auf logisch null(GND) gesetzt.

[0052] Das erste Logikelement in der Zählerschaltung ist der Taktgeber ("Logic clock" in **Fig.** 5). Dieses Element ist einfach ein Ein-Bit-Register **404** (**Fig.** 4), auf das mit Software geschrieben werden kann. Das Umschalten von Register **404** per Software-Steuerung ergibt den Takt Local clock für die Zählerschaltung. Das nächste Zählerschaltungselement ist ein Toggle-Flipflop wie z. B. der Flipflop **402** ("Logic tff" in **Fig.** 5). Dieser Flipflop ist so definiert, dass er einen von Westen kommenden Eingang hat. (Laut Standardnomenklatur im XC6200 Data Sheet bedeuten die Bezeichnungen Logic.EAST und Ein ein nach Osten gehendes Signal, d. h. ein Signal, das von Westen kommt.) Das Toggle-Flipflop-Element bildet den Zustandsspeicher für den Zähler. Als Nächstes ist das Übertrags-Logikelement **401** für den Zähler ("Logic carry" in **Fig.** 5) einfach ein AND-Gate mit Eingängen von der Übertragslogik der vorherigen Stufe und dem Ausgang des Toggle-Flipflop der aktuellen Stufe. Das Übertragselement erzeugt das "Toggle"-Signal für die nächste Stufe des Zählers. Die **Fig.** 6A und 6B stellen jeweils die Flipflop- bzw. Übertragslogikzelle in einem XC6200-Bauelement grafisch dar. Schließlich wird eine Logisch-"Eins" oder VCC-Zelle ("Logic one" in **Fig.** 5, Block **403** in **Fig.** 4) für den Übertragseingang zur ersten Stufe des Zählers implementiert.

[0053] Nach dem Definieren dieser Sammlung von abstrakten Elementen können diese an einer beliebigen Stelle in der XC6200-Zellenarray instanziiert werden. Diese Instanziierung erfolgt dadurch, dass ein Ruf an die mit jedem Objekt assoziierte write() Funktion erfolgt. Diese Funktion nimmt einen Spalten- und Reihenparameter, der die Zelle in dem zu konfigurierenden XC6200-Bauelement definiert. Zusätzlich wird das Hardware-Schnittstellenobjekt als Parameter geleitet. In diesem Fall erfolgt die gesamte Konfiguration zu pci6200, einer einzelnen XC6200DS PCI-Karte.

[0054] Ein Beispiel für diese Instanziierung ist in **Fig.** 7 dargestellt, wo die Elemente für den Zähler von **Fig.** 4 instanziiert werden. Der Code in **Fig.** 7 führt die gesamte notwendige Konfiguration durch. In der for() Schleife befinden sich die Übertragszellen (**401** in **Fig.** 4) in einer Spalte mit den Toggle-Flipflops tff (**402** in **Fig.** 4) in der nächsten Spalte. Ein lokales Takt- und ein Löschsignal hängen an jedem Toggle-Flipflop tff an. Der relative Ort dieser Zellen ist in **Fig.** 4 dargestellt.

[0055] Unter der for( ) Schleife wird eine Konstante "1" als Eingang zur Übertragskette (**403** in **Fig.** 4) ge-

setzt. Als Nächstes wird der Software-gesteuerte Taktgeber (Local_clock in **Fig.** 4) konfiguriert. Dies ist das Taktobjekt, bei dem das localClock-Routing an die Toggle-Flipflops tff des Zählers angehängt ist. Schließlich wird der globale Takt zum Synchronisieren des Software-gesteuerten lokalen Taktgebers verwendet. In einigen Ausgestaltungen werden die Takt- und Lösch-Grundelemente nicht benötigt; in dieser Ausgestaltung ist ihre Anwesenheit zum Unterstützen der XC6200-Architektur notwendig.

[0056] Nach dem Konfigurieren der Schaltung ist es eine einfache Sache, die Register-Objekte jeweils über die Funktionen get() bzw. set() zu lesen und zu schreiben. In **Fig.** 8A wird der Takt durch abwechselndes Schreiben von "0" und "1" auf das Taktregister (**404** in

[0057] **Fig.** 4) umgeschaltet. Das Zählerregister (nicht dargestellt) dient zum Lesen des Wertes des Zählers (Ausgänge COUNT[0], COUNT[1], COUNT[2], usw.). **Fig.** 8B zeigt den Ablauf der Ausführung dieses Codes, der auf dem XC6200DS-Entwicklungssystem läuft.

Schlussfolgerungen

[0058] Dies ist zwar nur ein einfaches Beispiel für Demonstrationszwecke, aber es werden darin alle MDRC-Merkmale genutzt. Diese Merkmale beinhalten Registerlese- und – schreibvorgänge sowie Merkmale wie Software-gesteuertes lokales Takten. Weitere komplexere Schaltungen wurden ebenfalls mit MDRC entwickelt. Komplexere Schaltungen werden mit denselben Grundmerkmalen aufgebaut; der Hauptunterschied besteht in der Größe des Codes.

[0059] MDRC bietet ein einfaches, schnelles, integriertes Tool für eine umkonfigurierbare logikgestützte Verarbeitung. MDRC ist derzeit ein manuelles Tool, da der Programmierer Platzierung und Routing von Schaltungen für umkonfigurierbares Rechnen streng kontrollieren sollte. MDRC bietet jedoch sehr schnelle Kompilationszeiten im Austausch für den manuellen Design-Stil. Die zum Erzeugen dieser Schaltungen und des Laufzeit-Supportcode notwendigen Kompilationszeiten liegen in der Größenordnung von Sekunden, viele Größenordnungen schneller als die Design-Zykluszeit herkömmlicher CAD-Tools. Diese ungewöhnliche Geschwindigkeit lässt eine Entwicklung in einer Umgebung zu, die einer modernen integrierten Software-Entwicklungsumgebung ähnlich ist. Darüber hinaus lässt die objektorientierte Natur von Java die Erstellung von Bibliotheken von parameterisierten Zellen zu. Dieses Merkmal könnte die Produktivität von MDRC-Benutzern stark erhöhen.

[0060] MDRC kann als Basis für ein traditionelles grafisches CAD-Tool verwendet werden. Dieser Ansatz wäre zum Produzieren von statischen Schaltungen nützlich. Der obige Text beschreibt MDRC im Zusammenhang mit FPGAs, die für dynamisch umkonfigurierbares Rechnen verwendet werden, wie z. B. die FPGA-Familie Xilinx XC6200. Die Erfindung kann jedoch auch auf andere FPGAs und andere mit Software programmierbare ICs angewendet werden, die nicht für dynamisch umkonfigurierbares Rechnen verwendet werden.

[0061] Für Fachpersonen in den jeweiligen Bereichen der Erfindung werden verschiedene Modifikationen und Zusätze offensichtlich sein, die sich aus der vorliegenden Offenbarung ergeben. Demgemäß werden alle solche Modifikationen und Zusätze als in den Umfang der Erfindung fallend angesehen.

Patentansprüche

1. Verfahren zum Konfigurieren einer frei programmierbaren logischen Anordnung FPGA (**106**) zum dynamisch umkonfigurierbaren Rechnen, wobei das Verfahren die folgenden Schritte umfasst:
a) Programmieren des Host-Prozessors mit Anweisungen (**201**) in einer höheren Programmiersprache;
b) Bereitstellen eines Compilers (**203**) für die höhere Programmiersprache, der auf dem Host-Prozessor läuft, um ablauffähigen Code (**204**) als Reaktion auf die Anweisungen (**201**) zu erzeugen, wobei der ablauffähige Code (**204**) kompilierte Platzierungs- und Leitweginformationen beinhaltet; und
c) Verbinden des Host-Prozessors mit der FPGA (**106**) für eine dynamische Umkonfigurationsprogrammierung der FPGA (**106**) durch den Host-Prozessor über den ablauffähigen Code (**204**).

2. Verfahren nach Anspruch 1, bei dem die höhere Programmiersprache Java ist.

3. Verfahren nach Anspruch 1 oder 2, ferner umfassend den folgenden Schritt:
d) Instanziieren von Elementen von einer Bibliothek (**202**) von Elementen, die mit dem Compiler (**203**) kompatibel sind.

4. Verfahren nach Anspruch 3, bei dem die Bibliothek (**202**) kombinatorische Logikelemente (**401**) umfasst.

5. Verfahren nach Anspruch 3, bei dem die Bibliothek (**202**) Flipflop-Elemente (**402**, **404**) umfasst.

6. Verfahren nach Anspruch 3, bei dem die Bibliothek (**202**) Latch-Elemente (**401**) umfasst.

Es folgen 7 Blatt Zeichnungen

Anhängende Zeichnungen

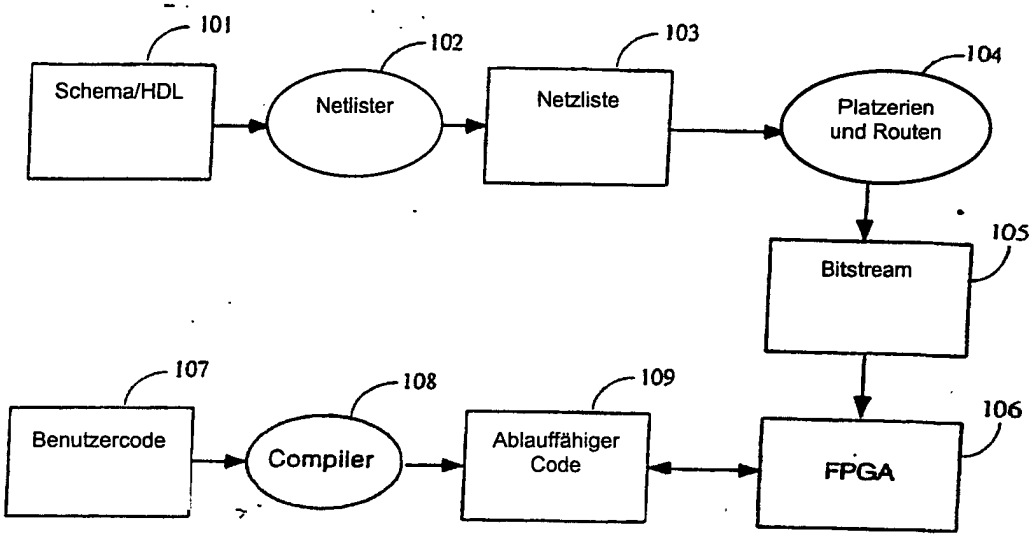

FIG. 1 **(Stand der Technik)**

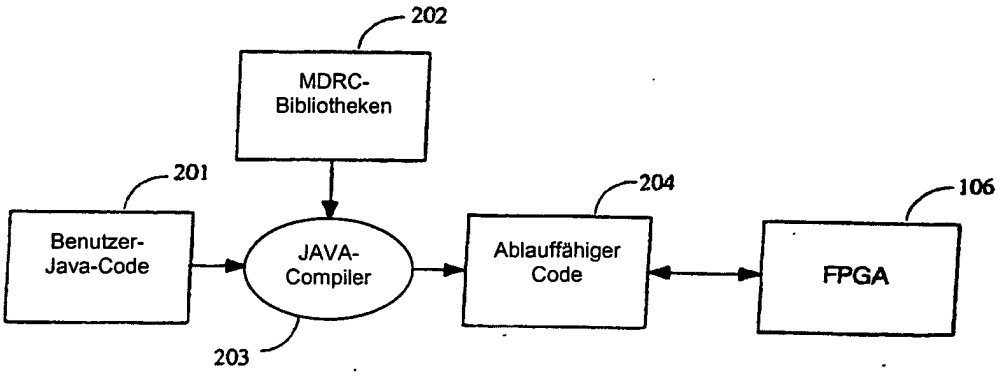

FIG. 2

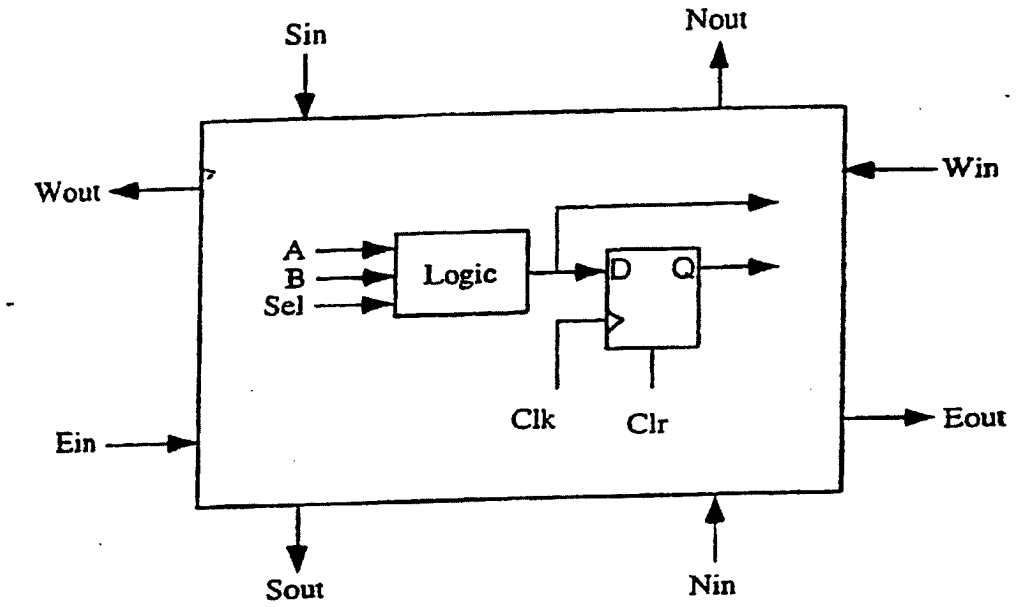

FIG. 3

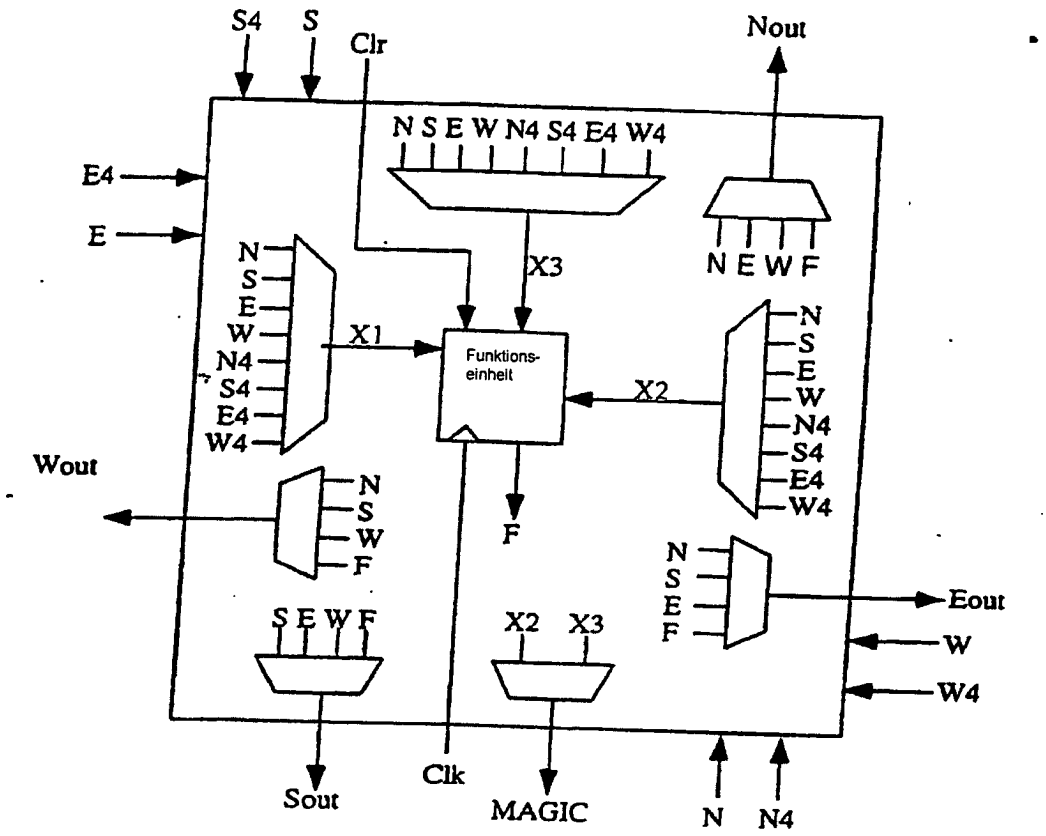

FIG. 3A
(Stand der Technik)

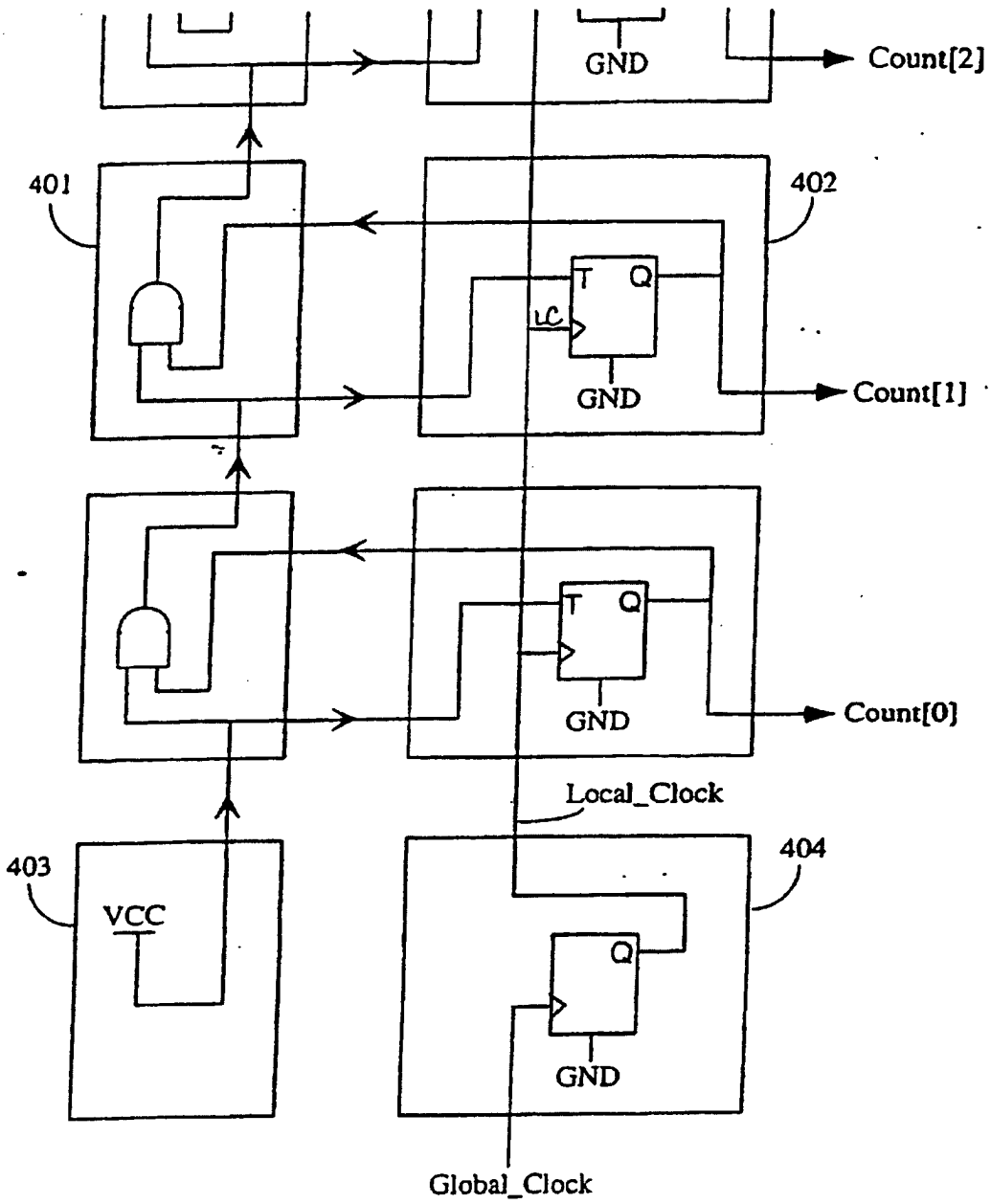
GND
Count[2]
401
402
T
Q
LC
GND
Count[1]
T
Q
GND
Count[0]
Local_Clock
403
VCC
404
Q
GND
Global_Clock

FIG. 4

```
Pci6200 pci6200 = new Pci6200N(null);  // Hardware interface
pci6200.connect();
Register counterReg =       new Register(COLUMN, counterMap, pci6200);
Register clockReg =         new Register(COLUMN, clockMap, pci6200);
ClockMux localClock =       new ClockMux(ClockMux.CLOCK_IN);
ClockMux globalClock =      new ClockMux(ClockMux.GLOBAL_CLOCK);
ClearMux clear =            new ClearMux(ClearMux.ZERO);
Logic tff =                 new Logic(Logic.T_FLIP_FLOP, Logic.EAST);
Logic clock =               new Logic(Logic.REGISTER);
Logic one =                 new Logic(Logic.ONE);
Logic carry =               new Logic(Logic.AND, Logic.NORTH, logic.WEST);
carry.setEastOutput(Logic.NORTH);  // Set carry output
```

FIG. 5

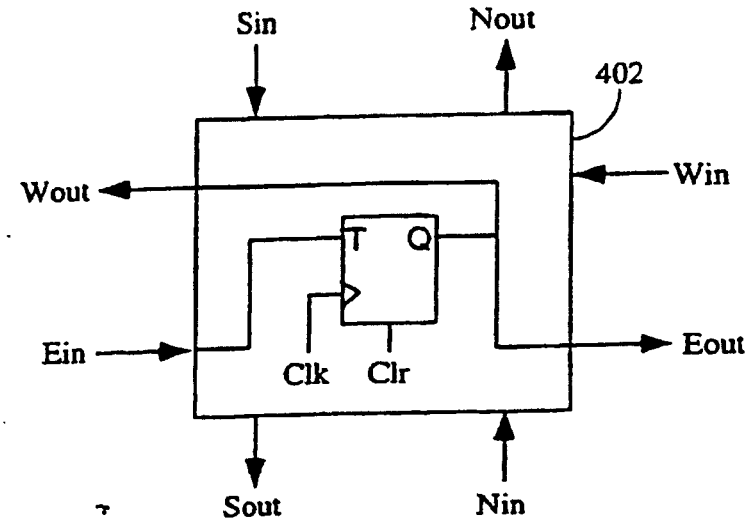
Sin
Nout
402
Win
Wout
T
Q
Ein
Eout
Clk
Clr
Sout
Nin

FIG. 6A

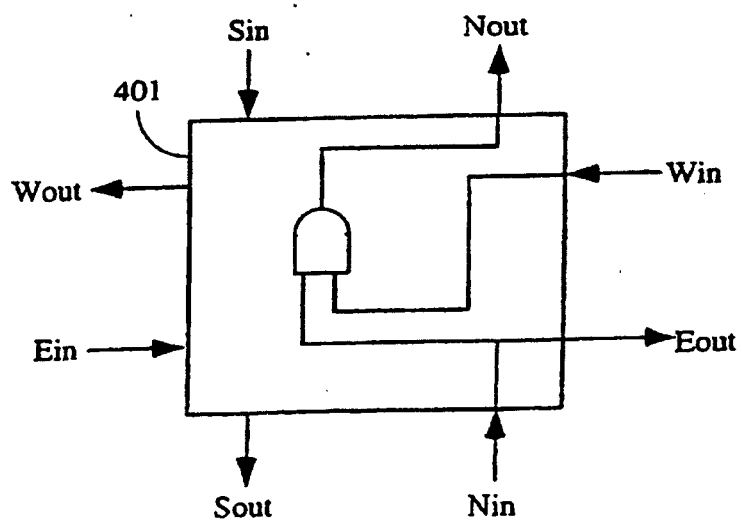
Sin
Nout
401
Win
Wout
Ein
Eout
Sout
Nin

FIG. 6B

```
/ * Configure cells */
for (i=ROW_START; i<ROW_END; i++) { // The counter
        carry.write((COLUMN-1), i, pci6200);
  tff.write(COLUMN, i, pci6200);
        localClock.write(COLUMN, i, pci6200);
      clear.write(COLUMN, i, pci6200);
    } /* end for */
one.write((COLUMN-1), (ROW_START-1), pci6200); // Carry in
clock.write(COLUMN, (ROW_START-1), pci6200); // Clock
localClock.set(ClockMux.NORTH_OUT);
localClock.write(COLUMN, ROW_START, pci6200);
globalClock.write(COLUMN, (ROW_START-1), pci6200);
```

FIG. 7

```
for (i=0; i<5; i++) {
        clockReg.set(0); // Toggle clock
  clockReg.set(1);
        System.out.println("Count: " + counterReg.get());
    } /*end for() */
```

FIG. 8A

```
C: \java\JERC> java Counter
Count: 0
Count: 1
Count: 2
Count: 3
Count: 4
C: \java\JERC>
```

FIG. 8B