ORIGINAL

MORRISON & FOERSTER LLP
KARL J. KRAMER (Bar No. 136433)
KKramer@mofo.com
DIANA LUO (Bar No. 233712)
dluo@mofo.com
755 Page Mill Road
Palo Alto, CA 94304-1018
Telephone: (650) 813-5600
Facsimile (650) 494-0792

Attorneys for Plaintiff
ALTERA CORPORATION

#14
Pd
SJ

FILED
DEC 2 3 2010
[illegible]
CLERK, U.S. DISTRICT COURT
NORTHERN DISTRICT OF CALIFORNIA
[illegible]

UNITED STATES DISTRICT COURT

NORTHERN DISTRICT OF CALIFORNIA

ALTERA CORPORATION Plaintiff, v. INTELLITECH CORPORATION Defendant.	Case No. CV 10-05866 JCS **COMPLAINT FOR PATENT INFRINGEMENT** **DEMAND FOR JURY TRIAL**

Plaintiff Altera Corporation ("Altera"), by and through its attorneys, alleges as follows:

PARTIES

1. Altera is a corporation organized under the laws of the State of Delaware with its principal place of business at 101 Innovation Drive, San Jose, California 95134. Altera regularly conducts business in the Northern District of California.

2. Upon information and belief, Defendant Intellitech Corporation ("Intellitech") is a corporation organized under the laws of the State of New Hampshire with its principal place of business at 69 Venture Drive, Dover, New Hampshire 03820. Intellitech regularly conducts business in the Northern District of California.

JURISDICTION AND VENUE

3. This is an action for patent infringement arising under the patent laws of the United States, Title 35 of the United States Code. Jurisdiction as to these claims is conferred on this Court by 28 U.S.C. §§ 1331 and 1338(a).

4. Upon information and belief, this Court has personal jurisdiction over Intellitech because Intellitech has sufficient contacts with this judicial district and Intellitech regularly conducts business within this judicial district. Upon information and belief, Intellitech directly distributes, offers for sale or license, sells or licenses, and advertises its products and services within the State of California and this judicial district.

5. Venue is proper in this judicial district under 28 U.S.C. §§ 1391 and 1400(b). Intellitech is a corporation that is subject to personal jurisdiction in this district.

INTRADISTRICT ASSIGNMENT

6. This is an Intellectual Property Action to be assigned on a district-wide basis pursuant to Civil Local Rule 3-2(c).

BACKGROUND

7. Altera is a preeminent supplier of programmable logic solutions, offering Field Programmable Gate Arrays ("FPGAs"), Complex Programmable Logic Devices ("CPLDs"), and

Application-Specific Integrated Circuit ("ASICs") in combination with software tools, intellectual property, and support to customers world-wide.

8. The pioneering work of Altera's scientists and engineers has been rewarded with many United States Patents, including the following: 6,421,812 ("the '812 Patent"), 5,563,592 ("the '592 Patent"), and 7,036,046 ("the '046 Patent").

9. Each of the inventors of the Altera patents assigned their patents to Altera, so that Altera is and at all times has been the sole owner of all right and title to the patents, including the right to recover damages for past and current infringement.

10. Intellitech develops and licenses integrated circuits ("ICs"), Intellectual Property ("IP"), and software for configuration, debug and test of electronic products including System-on-a-Chip, ICs, print circuit boards ("PCB") and electronic systems. Intellitech's products include the SystemBIST product for configuring, programming, and testing memories, PLDs and other programmable devices; the Eclipse software for developing and validating system configurations and tests; the Fast Access Controller ("FAC") product for use with 1149.1 test infrastructures to enable in-system programming of FLASH memory devices; and the NEBULA Silicon Debugger for debugging and validation of ICs. Primary end users of these products include electronic product manufacturers and the semiconductor industry.

COUNT I

(Infringement of the '812 Patent)

11. Altera hereby restates and realleges the allegations set forth in paragraphs 1 through 10 above and incorporates them by reference.

12. On July 16, 2002, the '812 Patent entitled "Programming mode selection with JTAG circuits" was duly and legally issued to Altera by the United States Patent and Trademark Office. Altera is the owner of the entire right, title, and interest in and to the '812 Patent. A true and correct copy of the '812 Patent is attached as Exhibit A to this Complaint.

13. Altera has not licensed or otherwise authorized Intellitech to make, use, offer for sale or sell any products that embody the inventions of the '812 Patent.

14. Intellitech has directly infringed and continues to directly infringe the '812 Patent by its unlicensed use of methods that embody the invention claimed by the '812 Patent in the United States during development, prototyping, testing, verification, and demonstration of its SystemBIST and FAC products with the Eclipse Software.

15. Intellitech has had actual knowledge of the '812 Patent since at least November 4, 2010.

16. Intellitech has indirectly infringed and continues to indirectly infringe the '812 Patent by inducing end users to infringe the '812 Patent by using the SystemBIST and FAC products with Eclipse Software. Intellitech intentionally took action that induced end users to infringe the '812 Patent by marketing, selling, and supporting the SystemBIST and FAC products and Eclipse Software. Intellitech had awareness of the '812 Patent in circumstances in which it knew or should have known that its actions would cause direct infringement by end users.

17. Intellitech has indirectly infringed and continues to indirectly infringe the '812 Patent by contributing to direct infringement by end users who use the SystemBIST and FAC products with Eclipse Software. Intellitech supplied an important component of the infringing part of the method, the component is not a common component suitable for non-infringing use, and Intellitech supplied the component with the knowledge of the '812 Patent and knowledge that the component was especially made or adapted for use in an infringing manner.

18. Upon information and belief, Intellitech's infringement of Altera's '812 Patent has been and will continue to be willful, wanton and deliberate.

19. Altera is damaged and irreparably injured by Intellitech's infringing activities and will continue to be so damaged and irreparably injured unless Intellitech's infringing activities are enjoined by this Court.

20. Intellitech is thus liable to Altera for infringement of the '812 Patent pursuant to 35 U.S.C. § 271.

COUNT II

(Infringement of the '592 Patent)

21. Altera hereby restates and realleges the allegations set forth in paragraphs 1 through 20 above and incorporates them by reference.

22. On October 8, 1996, the '592 Patent entitled "Programmable logic device having a compressed configuration file and associated decompression" was duly and legally issued to Altera by the United States Patent and Trademark Office. Altera is the owner of the entire right, title, and interest in and to the '592 Patent. A true and correct copy of the '592 Patent is attached as Exhibit B to this Complaint.

23. Altera has not licensed or otherwise authorized Intellitech to make, use, offer for sale or sell any products that embody the inventions of the '592 Patent.

24. Intellitech has directly infringed and continues to directly infringe the '592 Patent by its unlicensed use of methods that embody the invention claimed by the '592 Patent in the United States during development, prototyping, testing, verification, and demonstration of the SystemBIST product with Eclipse Software to program FLASH devices.

25. Intellitech has had actual knowledge of the '592 Patent since at least November 4, 2010.

26. Intellitech has indirectly infringed and continues to indirectly infringe the '592 Patent by inducing end users to directly infringe the '592 Patent by using the SystemBIST and FAC products with Eclipse Software. Intellitech intentionally took action that induced end users to infringe the '592 Patent by marketing, selling, and supporting the SystemBIST and FAC products. Intellitech had awareness of the '592 Patent in circumstances in which it knew or should have known that its actions would cause direct infringement by end users.

27. Intellitech has indirectly infringed and continues to indirectly infringe the '592 Patent by contributing to direct infringement by end users who use the SystemBIST and FAC products. Intellitech supplied an important component of the infringing part of the method, the component is not a common component suitable for non-infringing use, and Intellitech supplied the component with the knowledge of the '592 Patent and knowledge that the component was especially made or adapted for use in an infringing manner.

28. Upon information and belief, Intellitech's infringement of Altera's '592 Patent has been and will continue to be willful, wanton and deliberate.

29. Altera is damaged and irreparably injured by Intellitech's infringing activities and will continue to be so damaged and irreparably injured unless Intellitech's infringing activities are enjoined by this Court.

30. Intellitech is thus liable to Altera for infringement of the '592 Patent pursuant to 35 U.S.C. § 271.

COUNT III

(Infringement of the '046 Patent)

31. Altera hereby restates and realleges the allegations set forth in paragraphs 1 through 50 above and incorporates them by reference.

32. On April 25, 2006, the '046 Patent entitled "PLD debugging hub" was duly and legally issued to Altera by the United States Patent and Trademark Office. Altera is the owner of the entire right, title, and interest in and to the '046 Patent. A true and correct copy of the '046 Patent is attached as Exhibit C to this Complaint.

33. Altera has not licensed or otherwise authorized Intellitech to make, use, offer for sale or sell any products that embody the inventions of the '046 Patent.

34. Intellitech has directly infringed and continues to directly infringe the '046 Patent by its unlicensed use of systems that embody the invention claimed by the '046 Patent in the

United States during development, prototyping, testing, verification, and demonstration of the NEBULA Silicon Debugger product.

35. Intellitech has had actual knowledge of the ’046 Patent since at least November 4, 2010.

36. Intellitech has indirectly infringed and continues to indirectly infringe the ’046 Patent by inducing end users to directly infringe the ’046 Patent by using the NEBULA Silicon Debugger. Intellitech intentionally took action to induce end users to infringe the ’046 Patent by marketing, selling, and supporting the NEBULA Silicon Debugger. Intellitech had awareness of the ’046 Patent in circumstances in which it knew or should have known that its actions would cause direct infringement by end users.

37. Intellitech has indirectly infringed and continues to indirectly infringe the ’046 Patent by contributing to direct infringement by end users who use the NEBULA Silicon Debugger. Intellitech supplied an important component of the infringing system, the component is not a common component suitable for non-infringing use, and Intellitech supplied the component with the knowledge of the ’046 Patent and knowledge that the component was especially made or adapted for use in an infringing manner.

38. Upon information and belief, Intellitech’s infringement of Altera’s ’046 Patent has been and will continue to be willful, wanton and deliberate.

39. Altera is damaged and irreparably injured by Intellitech’s infringing activities and will continue to be so damaged and irreparably injured unless Intellitech’s infringing activities are enjoined by this Court.

40. Intellitech is thus liable to Altera for infringement of the ’046 Patent pursuant to 35 U.S.C. § 271.

PRAYER FOR RELIEF

WHEREFORE, Altera prays for judgment as follows:

A. Entry of judgment holding Intellitech liable for infringement of the patents at issue in this litigation;

B. An order permanently enjoining Intellitech, its officers, agents, servants, employees, attorneys and affiliated companies, its assigns and successors in interest, and those persons in active concert or participation with it, from continued acts of infringement of the patents at issue in this litigation;

D. An order awarding Altera statutory damages and damages according to proof resulting from Intellitech's infringement of the patents at issue in this litigation, together with prejudgment and post-judgment interest;

E. Trebling of damages under 35 U.S.C. § 284 in view of the willful and deliberate nature of Intellitech's infringement of the patents at issue in this litigation;

F. An order awarding Altera its costs and attorney's fees under 35 U.S.C. § 285; and

G. Any and all other legal and equitable relief as may be available under law and which the court may deem proper.

Dated: December 23, 2010

KARL J. KRAMER
DIANA LUO
MORRISON & FOERSTER LLP

By: [illegible]
KARL J. KRAMER

Attorneys for Plaintiff
ALTERA CORPORATION

DEMAND FOR A JURY TRIAL

Plaintiff hereby demands a jury trial on all issues so triable under the laws as provide by Rule 38(b) of the Federal Rules of Civil Procedure.

Dated: December 23, 2010

KARL J. KRAMER
DIANA LUO
MORRISON & FOERSTER LLP

By: [illegible]
KARL J. KRAMER

Attorneys for Plaintiff
ALTERA CORPORATION

Exhibit A

US006421812B1

(12) **United States Patent**
Wang et al.

(10) **Patent No.:** **US 6,421,812 B1**
(45) **Date of Patent:** **Jul. 16, 2002**

(54) PROGRAMMING MODE SELECTION WITH JTAG CIRCUITS

(75) Inventors: **Xiaobao Wang**, Santa Clara; **Chiakang Sung**, Milpitas; **Joseph Huang**, San Jose; **Bonnie Wang**, Cupertino; **Khai Nguyen**, San Jose; **Richard G. Cliff**, Milpitas, all of CA (US)

(73) Assignee: **Altera Corporation**, San Jose, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/094,186**

(22) Filed: **Jun. 9, 1998**

Related U.S. Application Data

(60) Provisional application No. 60/049,478, filed on Jun. 13, 1997, provisional application No. 60/050,953, filed on Jun. 13, 1997, provisional application No. 60/049,275, filed on Jun. 10, 1997, provisional application No. 60/049,246, filed on Jun. 10, 1997, provisional application No. 60/052,990, filed on Jun. 10, 1997, provisional application No. 60/049,247, filed on Jun. 10, 1997, provisional application No. 60/049,243, filed on Jun. 10, 1997, and provisional application No. 60/049,245, filed on Jun. 10, 1997.

(51) **Int. Cl.**^{7} **G06F 17/50**
(52) **U.S. Cl.** **716/5**
(58) **Field of Search** 716/5; 711/5; 714/727

(56) **References Cited**

U.S. PATENT DOCUMENTS

3,761,695	A	9/1973	Eichelberger	235/153 AC
3,783,254	A	1/1974	Eichelberger	235/152
3,806,891	A	4/1974	Eichelberger et al.	340/172.5
4,488,259	A	12/1984	Mercy	364/900
4,667,325	A	5/1987	Kitano et al.	371/25
4,701,920	A	10/1987	Resnick et al.	371/25
5,175,859	A	12/1992	Miller et al.	395/800
5,336,951	A	8/1994	Josephson et al.	307/465
5,355,369	A	10/1994	Greenbergerl et al.	371/22.3

(List continued on next page.)

FOREIGN PATENT DOCUMENTS

EP	0 639 006 A1	2/1995	
EP	0 828 163 A1	3/1998	
WO	WO 97/06599	2/1997	H03K/19/177

OTHER PUBLICATIONS

Altera Corporation, Data Sheet, "Flex 10K Embedded Programmable Logic Family," Jul., 1995, ver. 1, pp. 1–39.

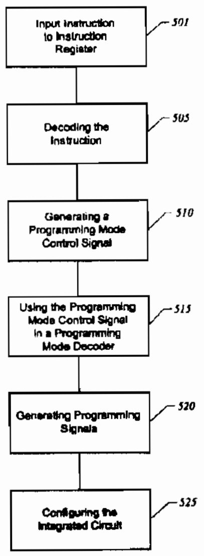
(List continued on next page.)

Primary Examiner—Matthew Smith
Assistant Examiner—Leigh Garbowski
(74) *Attorney, Agent, or Firm*—Townsend and Townsend and Crew LLP

(57) **ABSTRACT**

A technique to provide higher system performance by increasing amount of data that may be transferred in parallel is to increase the number of external pins available for the input and output of user data (user I/O). Specifically, a technique is to reduce the number of dedicated pins used for user I/O, leaving more external pins available for user I/O. The dedicated pins used to implement a function such as the JTAG boundary scan architecture may be also be used to provide other functionality, such as to select the programming modes. In a specific embodiment, a JTAG instruction code that is not already used for a JTAG boundary scan instruction stored in an instruction register (**220**) may be used to replace the programming mode select pins (**252**) in a programmable logic device (PLD).

29 Claims, 8 Drawing Sheets


Input Instruction to Instruction Register
501
Decoding the Instruction
505
Generating a Programming Mode Control Signal
510
Using the Programming Mode Control Signal in a Programming Mode Decoder
515
Generating Programming Signals
520
Configuring the Integrated Circuit
525

U.S. PATENT DOCUMENTS

5,361,373 A		11/1994	Gilson	395/800
5,489,858 A		2/1996	Pierce et al.	326/56
5,491,666 A		2/1996	Sturges	365/201
5,581,564 A		12/1996	Miller et al.	371/22.3
5,590,305 A		12/1996	Terrill et al.	395/430
5,594,367 A		1/1997	Trimberger et al.	326/41
5,644,496 A		7/1997	Agrawal et al.	364/489
5,650,734 A		7/1997	Chu et al.	326/38
5,734,868 A		3/1998	Curd et al.	395/500
5,737,567 A		4/1998	Whittaker et al.	395/430
5,829,007 A	*	10/1998	Wise et al.	711/5
5,841,867 A		11/1998	Jacobson et al.	380/25
5,869,979 A		2/1999	Bocchino	326/38
5,991,908 A	*	11/1999	Baxter et al.	714/727
6,058,255 A	*	5/2000	Jordan	716/5

OTHER PUBLICATIONS

Altera Corporation, Data Sheet, "Flex 8000 Programmable Logic Device Family," Aug., 1994, ver. 4, pp. 1–22.

Altera Corporation, Data Sheet, "Max 7000 Programmable Logic Device Family," Jun. 1996, ver 4, pp. 193–261.

Altera Corporation, Application Note 39, "JTAG Boundary–ScanTesting In Altera Devices," Nov., 1995, ver. 3, pp. 1–28.

IEEE Computer Society, "IEEE Standard Test Access Port and Boundary–Scan Architecture (IEEE Std 1149.1–1990)," Institute of Electrical and Electronics Engineers, Inc., New York, NY, Oct. 21, 1993, pp. 1–1 to 12–6 and Appendix A–1 to A–12.

IEEE Computer Society, "Supplement to (IEEE Std 1149.1–1990), IEEE Standard Test Access Port and Boudary–Scan Architecture (IEEE Std 11493.1b–1994)," Institute of Electrical and Electronics Engineers, Inc., New York, NY, Mar. 1, 1995, pp. 1–67.

Xilnix Corporation, "The Programmable Logic Data Book," 1993, pp. 1–1 to 10–8.

Xilnix Corporation, "The Programmable Logic Data Book," Section 9, 1994, pp. 9–1 to 9–32.

Xilnix Corporation, "The Programmable Logic Data Book," Product Description, "XC2000 Logic Cell Array Families," Aug. 1994, pp. 2–187 to 2–216.

Xilnix Corporation, "The Programmable Logic Data Book," Product Description, "XC3000, XC3000A, XC000L, SC3100, XC3100A Logic Cell Array Families," pp. 2–105 to 2–152.

Xilnix Corporation, "The Programmable Logic Data Book," Product Specification, "XC4000 Series Field Programmable Gate Arrays," Jul. 30, 1996, version 1.03, pp. 4–5 to 4–76.

* cited by examiner

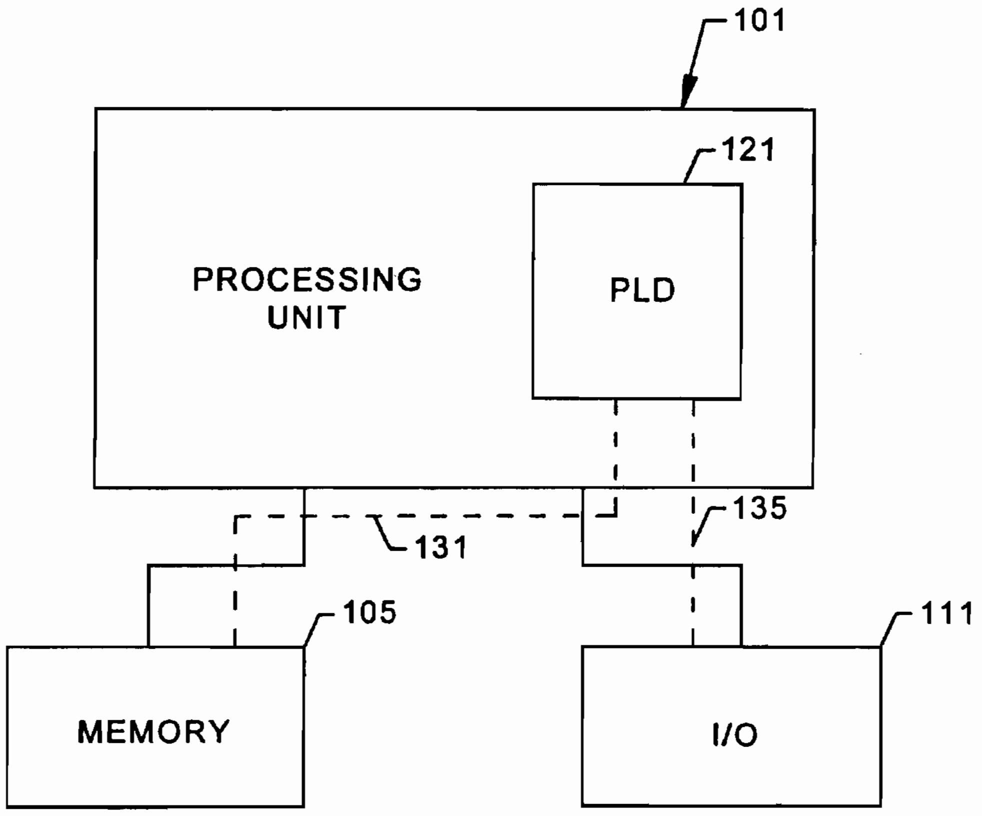


FIGURE 1

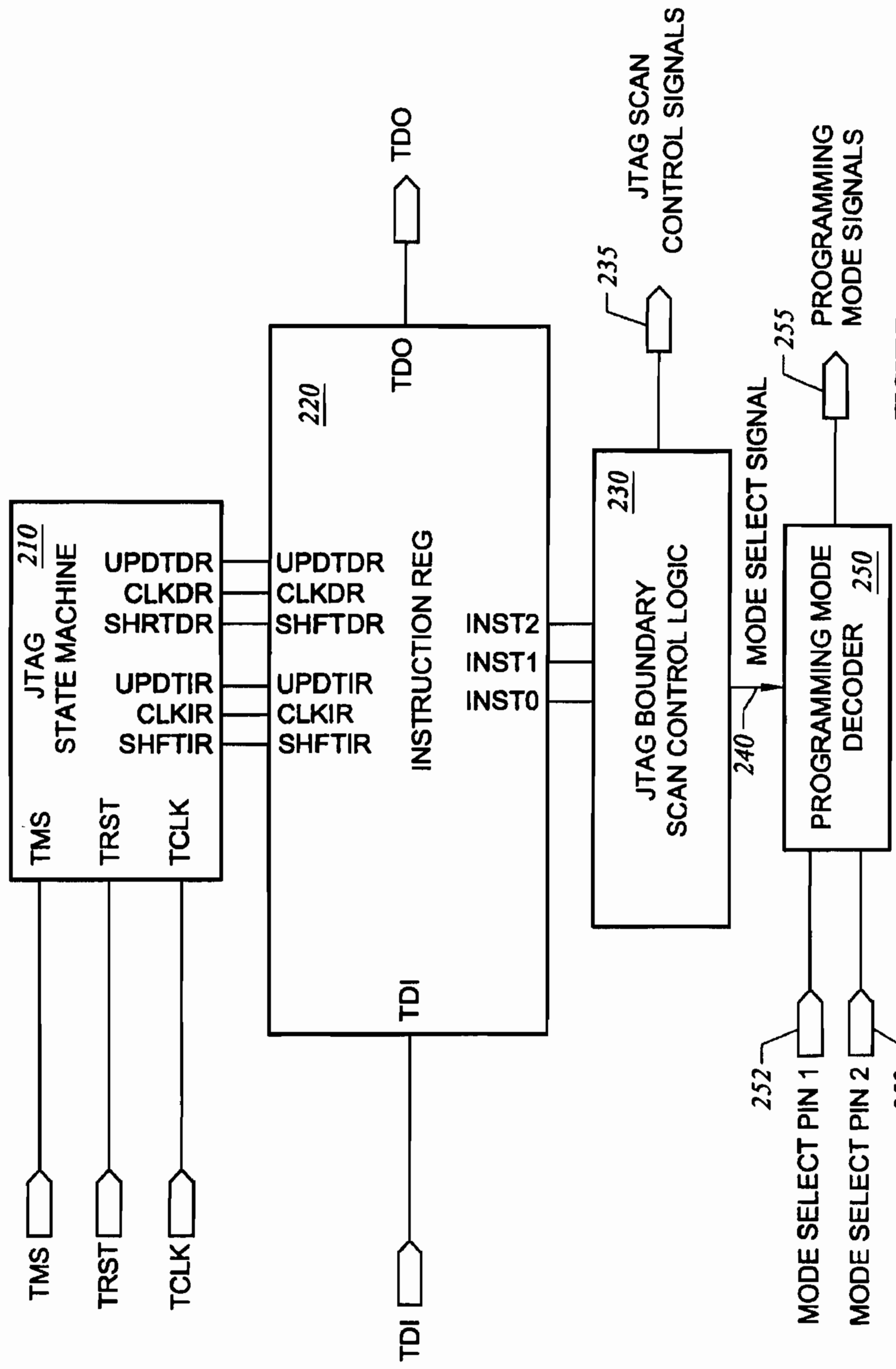

JTAG STATE MACHINE
210
UPDTDR
CLKDR
SHRTDR
UPDTIR
CLKIR
SHFTIR
TMS
TRST
TCLK
TMS
TRST
TCLK
220
INSTRUCTION REG
UPDTDR
CLKDR
SHFTDR
UPDTIR
CLKIR
SHFTIR
INST2
INST1
INST0
TDO
TDI
TDO
TDI
230
JTAG BOUNDARY SCAN CONTROL LOGIC
235
JTAG SCAN CONTROL SIGNALS
MODE SELECT SIGNAL
240
PROGRAMMING MODE DECODER 250
255
PROGRAMMING MODE SIGNALS
252
253
MODE SELECT PIN 1
MODE SELECT PIN 2

FIGURE 2

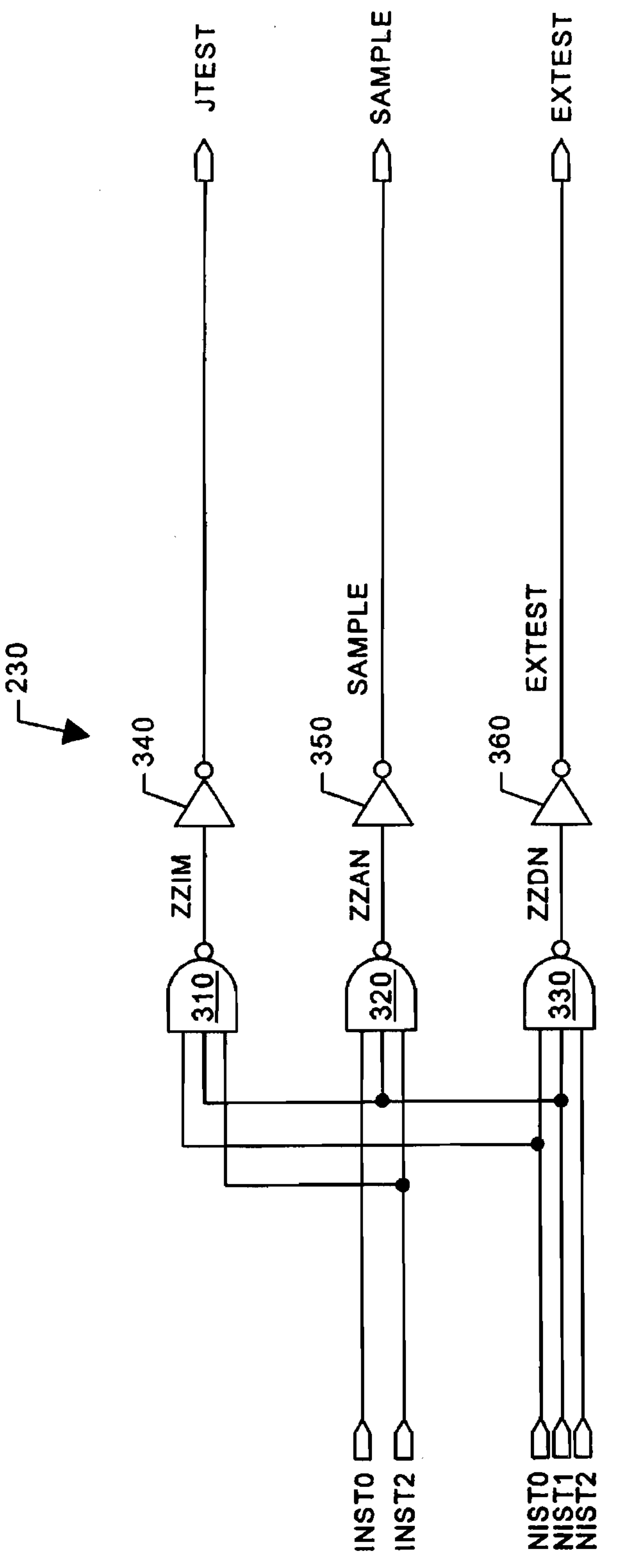

230
JTEST
SAMPLE
EXTEST
340
350
360
SAMPLE
EXTEST
ZZIM
ZZAN
ZZDN
310
320
330
INST0
INST2
NIST0
NIST1
NIST2

FIGURE 3

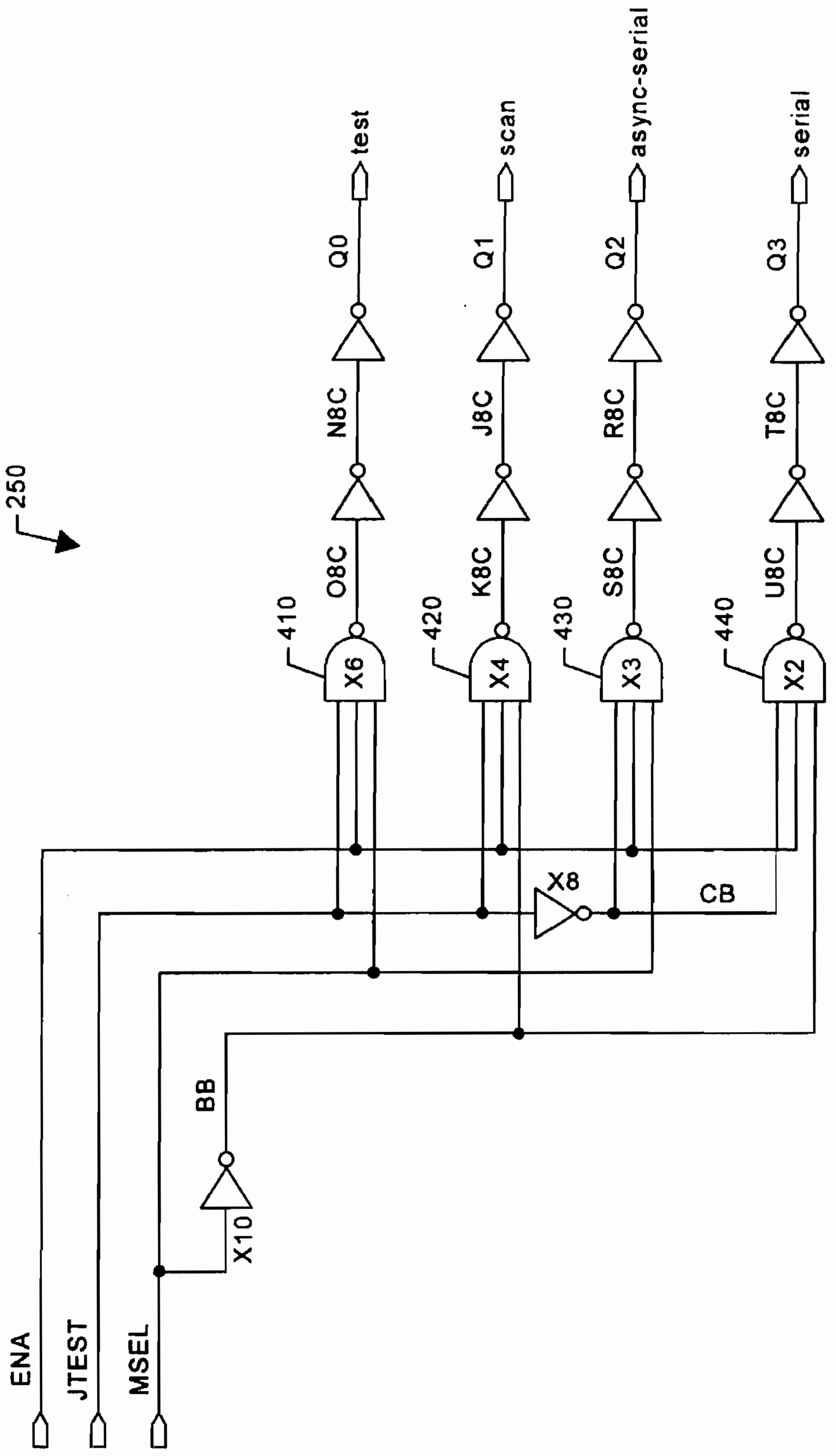

250
ENA
JTEST
MSEL
X10
BB
X8
CB
410
420
430
440
X6
X4
X3
X2
O8C
K8C
S8C
U8C
N8C
J8C
R8C
T8C
Q0
Q1
Q2
Q3
test
scan
async-serial
serial

FIGURE 4

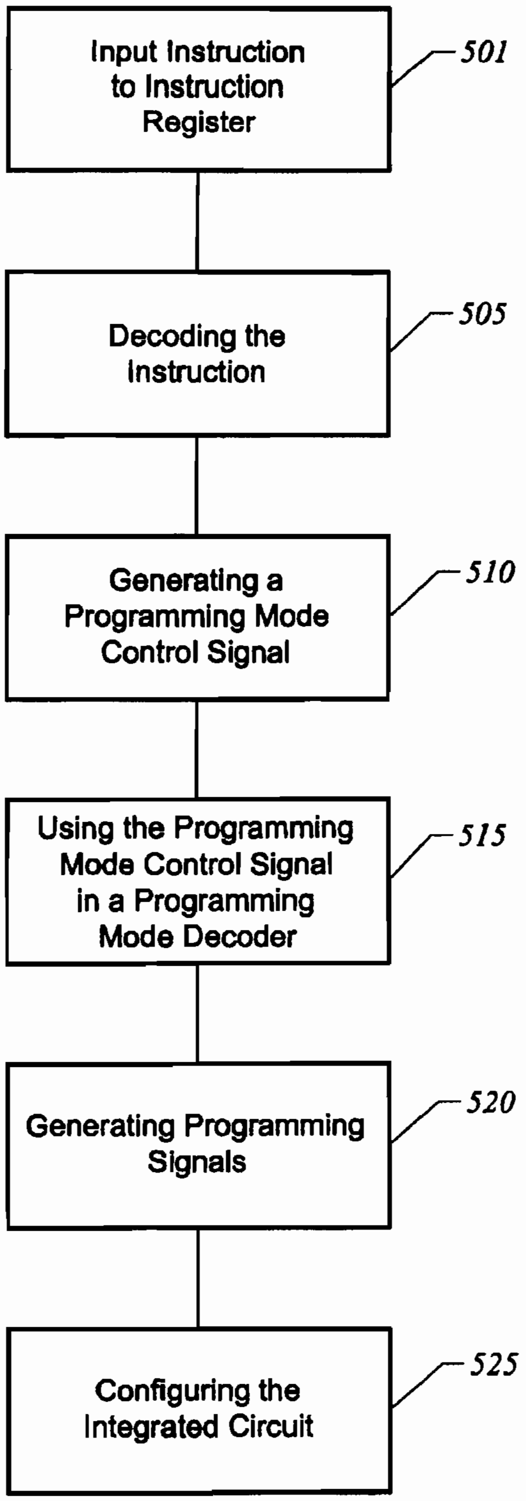

Input Instruction to Instruction Register
501
Decoding the Instruction
505
Generating a Programming Mode Control Signal
510
Using the Programming Mode Control Signal in a Programming Mode Decoder
515
Generating Programming Signals
520
Configuring the Integrated Circuit
525

FIGURE 5

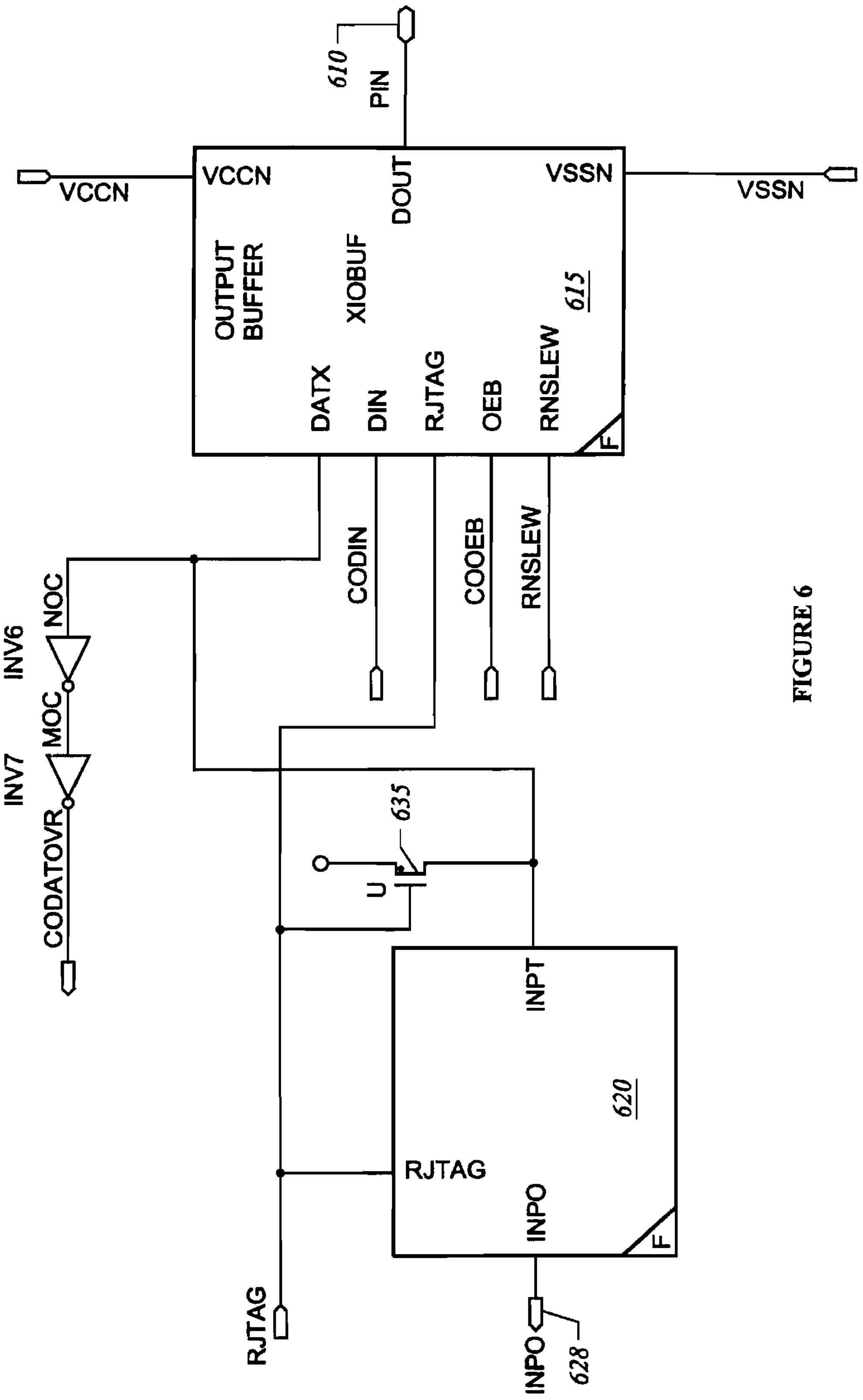

610
PIN
VCCN
VCCN
DOUT
VSSN
VSSN
OUTPUT BUFFER
XIOBUF
615
DATX
DIN
RJTAG
OEB
RNSLEW
F
CODIN
COOEB
RNSLEW
NOC
INV6
MOC
INV7
CODATOVR
635
U
INPT
620
RJTAG
INPO
F
RJTAG
INPO
628

FIGURE 6

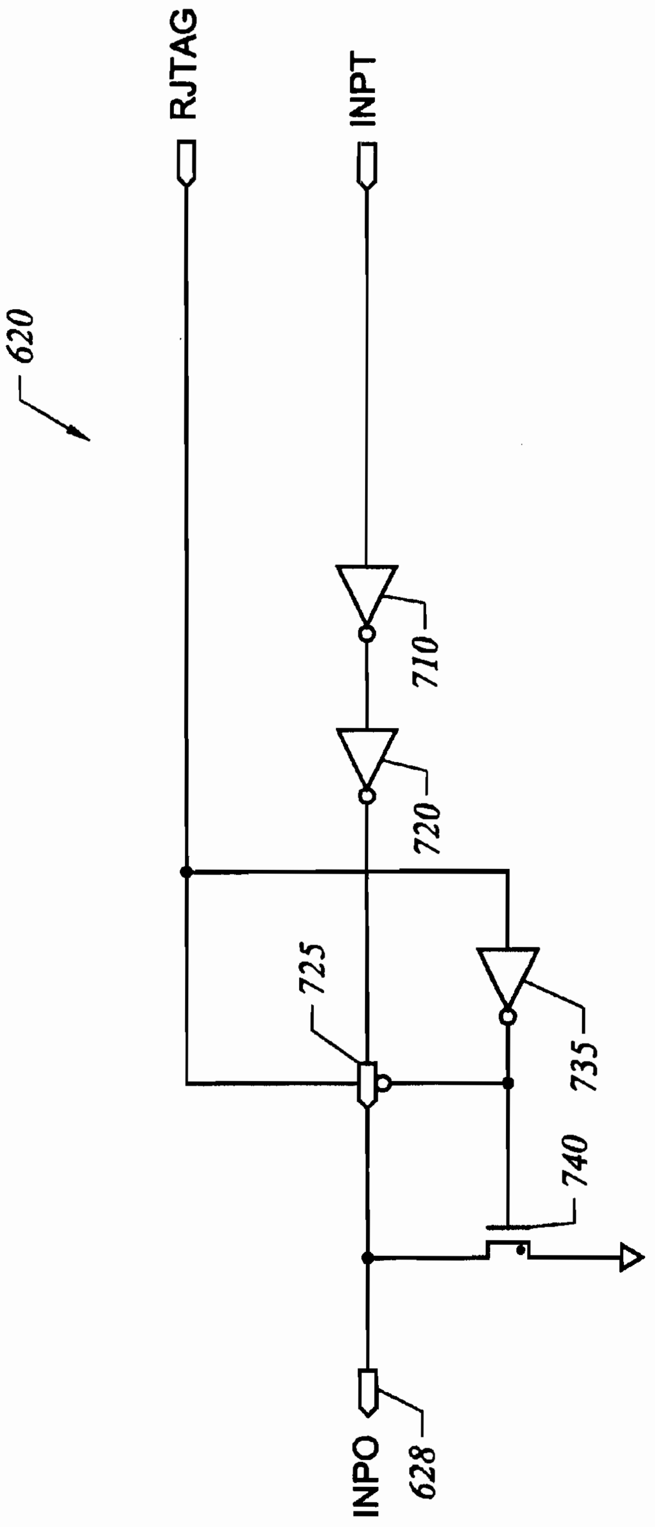


FIGURE 7

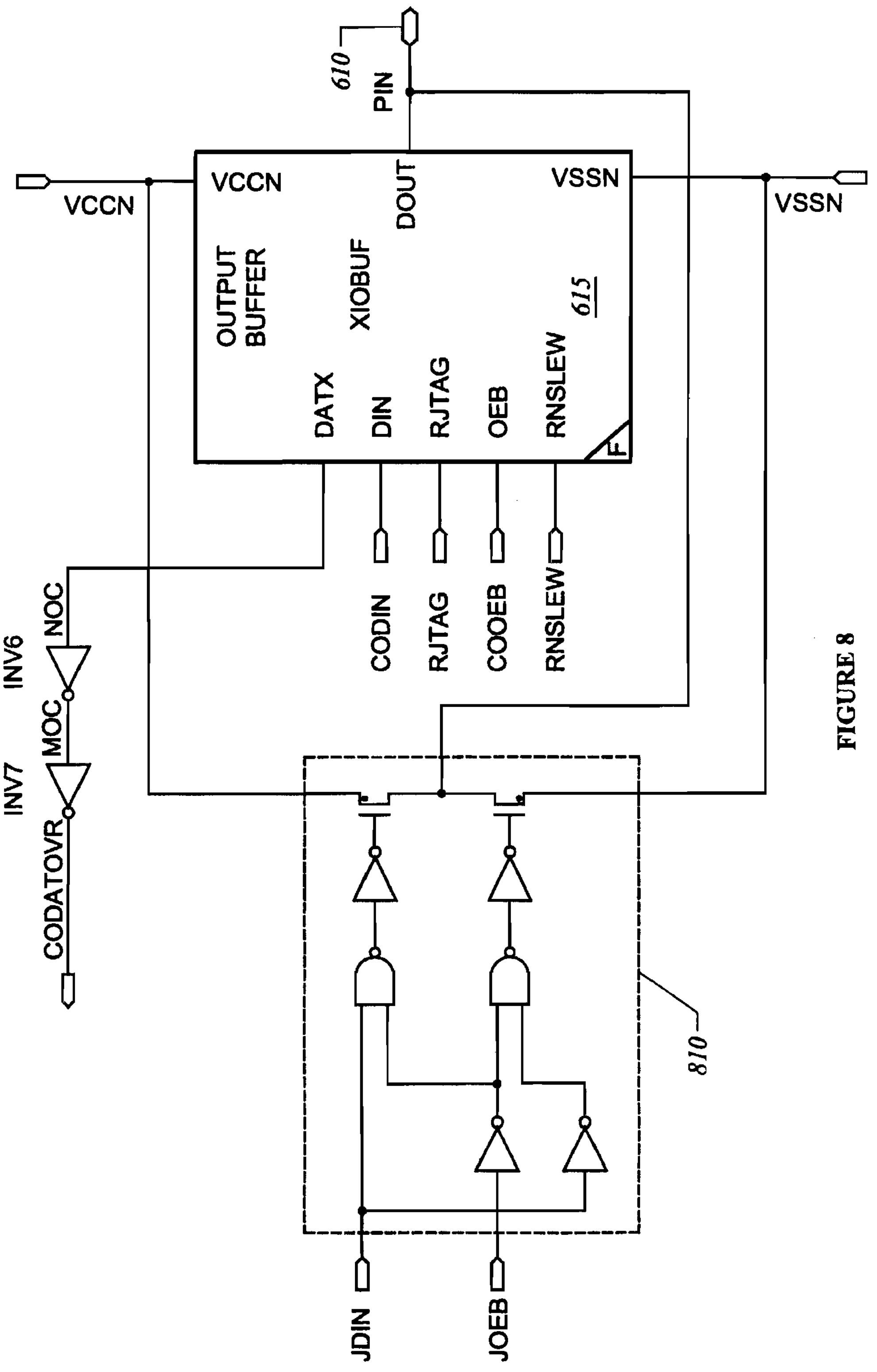

610
PIN
VCCN
VCCN
DOUT
VSSN
VSSN
OUTPUT BUFFER
XIOBUF
615
DATX
DIN
RJTAG
OEB
RNSLEW
F
NOC
INV6
MOC
INV7
CODATOVR
CODIN
RJTAG
COOEB
RNSLEW
810
JDIN
JOEB

FIGURE 8

PROGRAMMING MODE SELECTION WITH JTAG CIRCUITS

This application claims the benefit of a U.S. provisional application No. 60/049,275, filed Jun. 10, 1997; No. 60/049,478, filed Jun. 13, 1997; No. 60/049,246, filed Jun. 10, 1997; No. 60/052,990, filed Jun. 10, 1997; No. 60/049,247, filed Jun. 10, 1997; No. 60/049,243, filed Jun. 10, 1997; No. 60/050,953, filed Jun. 13, 1997; and No. 60/049,245, filed Jun. 10, 1997, all of which are incorporated herein by reference.

BACKGROUND OF THE INVENTION

The present invention relates to the field of integrated circuits, and more specifically to techniques to effectively provide greater number of external pins for input and output of data.

Semiconductor technology continues to improve. This technology allows greater and greater functionality to be provided by a single integrated circuit or "chip." Signals are input to and output from the chip using external pins or pads. The chip interfaces to external circuitry, possibly on other chips, using the external pins.

The performance of a system depends in part on the amount or rate at which data can be transferred on and off the chip. This transfer rate may be referred to as the data bandwidth. One technique for increasing system performance is to provide more rapid transfer rates. This may be accomplished by improvements in process technology or circuit design. Another technique to increase system performance is to transfer a greater amount of data at one time (or "in parallel"). Therefore, for greater performance, it is important there are many external pins available for input and output of user data.

In an integrated circuit, certain pins are sometimes dedicated to functions other than user data I/O. For example, in a programmable integrated circuit such as a PLD or FPGA, some pins may be dedicated to the programming and testing (such as JTAG boundary scan testing) of the device. These dedicated external pins reduce the number of pins available for user I/O. The performance of the chip may be detrimentally affected since not as many user I/O signals may be transferred in parallel.

Consequently, there is a need for techniques of effectively providing greater number of external pins for input and output to obtain higher performance. Specifically, there is a need for techniques to reduce the number of external pins dedicated to functions other than user I/O, which would make a greater number of external pins available for the input and output of user data.

SUMMARY OF THE INVENTION

The present invention is a technique to provide higher system performance by increasing the amount of data that may be transferred in parallel by increasing the number of external pins available for the input and output of user data (user I/O). One technique is to reduce the number of dedicated pins used for functions other than user I/O, leaving more external pins available for user I/O. The dedicated pins used to implement a function such as the JTAG boundary scan architecture may be also be used to provide other functionality, such as to select the programming modes. In a specific embodiment, a JTAG instruction code that is not already used for a JTAG boundary scan instruction may be used to replace the programming mode select pins in a programmable logic device (PLD).

In a technique of the present invention, the JTAG instruction used to replace the mode pins is shifted into a JTAG instruction register as are regular JTAG instructions. A JTAG boundary scan control logic block generates control signals to a programming mode decoder. Based on the instruction, the programming mode decoder selects the proper programming mode, and generates the appropriate programming mode signals. The programming mode signals are provided to the programming circuitry, and integrated circuit will be appropriately configured.

In a specific implementation, each single bit of the JTAG instruction code may be used to replace one programming mode select pin. In another implementation, the whole JTAG instruction code may be used to replace one mode select pin after instruction decoding. Technically, by doing this, many, or all, the mode pins can be eliminated, thus increasing the number of total available I/O pins. This concept provides advantages compared to JTAG programming and in-system programming (ISP) in such a way that a PLD device may be configured for different modes including test, scan, and programming modes.

The advantages of using JTAG instructions to replace programming the mode select pins of a programmable integrated circuit include saving device package costs and leaving space for more user I/Os. Overall, this increases the available functionality and value of the devices. There is relatively little cost to implement the circuits to implement PLD programming mode selection with JTAG circuits.

In a specific embodiment, the present invention is a method of configuring a programmable integrated circuit. An instruction is provided to a JTAG instruction register. The instruction is passed to a JTAG boundary scan control logic block. The JTAG boundary scan control logic block generates a control signal. The control signal is passed to a programming mode decoder. Based on the control signal, a programming mode signal is generated to place the programmable integrated circuit in a configuration mode.

Further, the present invention is a programmable integrated circuit including a JTAG state machine; an instruction register coupled to the JTAG state machine; a JTAG boundary scan control logic block coupled to the instruction register; and a programming mode decoder coupled to receive a mode signal from the JTAG boundary scan control logic block.

Another aspect of the present invention includes the use of JTAG circuitry resident on a programmable integrated circuit to select a programming mode of the integrated circuit. Further, the present invention includes the use of an instruction input to a JTAG instruction register, where this instruction is not used to perform a IEEE 1149.1 standard function, to place a programmable integrated circuit into a specific programming mode identified by the instruction. A still further aspect of the present invention is the use of JTAG circuits on a programmable logic device to place the programmable logic device in a configuration mode.

Other objects, features, and advantages of the present invention will become apparent upon consideration of the following detailed description and the accompanying drawings, in which like reference designations represent like features throughout the figures.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 shows a digital system incorporating a programmable logic device;

FIG. 2 shows an implementation of programming mode selection using JTAG circuitry;

FIG. 3 shows an implementation of JTAG boundary scan control logic circuitry;

FIG. 4 shows an implementation of a programming mode decoder;

FIG. 5 shows a flow diagram of a technique for configuring an integrated circuit;

FIG. 6 shows circuitry for selectably enabling use of a JTAG input pin;

FIG. 7 shows a circuit implementation of a JTAG input buffer; and

FIG. 8 shows a circuit implementation for selectably enabling use of a JTAG output pin.

DESCRIPTION OF THE SPECIFIC EMBODIMENTS

FIG. 1 shows a block diagram of a digital system. The system may be provided on a single board, on multiple boards, or even within multiple enclosures linked by electrical conductors or a network (e.g., a local area network or the internet). This digital system may be used in a wide variety of applications and industries including networking, telecommunications, automotive, control systems, consumer electronics, computers, workstations, military, industrial, digital processing, and many others. In the embodiment of FIG. 1, a processing unit **101** is coupled to a memory **105** and an I/O **111**. Further, a programmable logic device (PLD) **121** is incorporated within this digital system. PLD **121** may be specially coupled to memory **105** through connection **131** and to I/O **111** through connection **135**.

Programmable logic devices (PLDs) are sometimes also referred to as PALs, PLAs, FPLAS, CPLDs, EPLDs, EEPLDs, LCAs, or FPGAs. PLDs are well-known integrated circuits that provide the advantages of fixed integrated circuits with the flexibility of custom integrated circuits. Such devices allow a user to electrically program standard, off-the-shelf logic elements to meet a user's specific needs. See, for example, U.S. Pat. No. 4,617,479, incorporated herein by reference for all purposes. Such devices are currently represented by, for example, Altera's MAX® and FLEX® series of devices. The former are described in, for example, U.S. Pat. Nos. 5,241,224 and 4,871,930, and the *Altera Data Book,* June 1996, all incorporated herein by reference in their entirety for all purposes. The latter are described in, for example, U.S. Pat. Nos. 5,258,668, 5,260,610, 5,260,611, and 5,436,575, and the *Altera Data Book,* June 1996, all incorporated herein by reference in their entirety for all purposes.

Processing unit **101** may direct data to an appropriate system component for processing or storage, execute a program stored in memory **105** or input using I/O **111**, or other similar function. Processing unit **101** may be a central processing unit (CPU), microprocessor, floating point coprocessor, graphics coprocessor, hardware controller, microcontroller, programmable logic device programmed for use as a controller, or other processing unit. Memory **105** may be a random access memory (RAM), read only memory (ROM), fixed or flexible disk media, PC Card flash disk memory, tape, or any other storage retrieval means, or any combination of these storage retrieval means. PLD **121** may serve many different purposes within the system in FIG. **1**. PLD **121** may be a logical building block of processing unit **101**, supporting its internal and external operations. PLD **121** is programmed or configured to implement the logical functions necessary to carry on its particular role in system operation.

In a PLD, the number of available external pins limits the amount of data that may be input and output of the PLD at

the same time. The number of available external pins depends in part on the size and configuration of the package selected. Although larger package sizes provide greater numbers of external pins, it may not be desirable to use a larger package size since it will be more expensive, use more board space, and may have increased parasitics. Therefore, it is important to maximally use the available external pins for a given package.

On a typical PLD, some pins are dedicated for a particular purpose and other pins (i.e., I/O pins) are for input and output of logical data. For example, dedicated pins may be for testing or configuring the PLD. I/O pins are used to input and output user data. Dedicated pins cannot generally be used for user I/O. Therefore, the number of dedicated pins reduces the pins available for user I/O. Saving even a single dedicated pin, and using this pin instead for user I/O, may provide a great cost savings. For example, saving a single dedicated pin may avoid the use of the next larger package size.

In a PLD, there are typically dedicated pins for configuration and for testing. Configuration signals (e.g., pattern information) are input via a first set of dedicated pins. Test instructions and data (e.g., JTAG information) are input via a second set of dedicated pins. A technique to reduce the number of dedicated pins is to share the dedicated pins used for configuration and testing. The same amount of functionality would then be obtained using fewer dedicated pins This would increase the available number pins for user I/O. Although described with respect to PLDs, the techniques of the present invention are also applicable to other types of integrated circuits such as ASICs, microprocessors, and memories where it is desirable to reduce the number of dedicated pins and instead use these pins for user I/O.

FIG. 2 shows a specific embodiment of the present invention. FIG. 2 shows a block diagram of JTAG circuitry and programming mode selection circuitry. JTAG circuitry is discussed in some detail in Altera's Application Note 39, "IEEE 1149.1 (JTAG) Boundary-Scan Testing in Altera Devices," November 1995, incorporated herein by reference. In brief, the JTAG circuitry implements the IEEE 1149.1 specification or boundary-scan testing architecture. The JTAG circuitry can test pin connections without using physical test probes and can capture functional data while a device is operating normally.

The circuitry has JTAG dedicated pins TMS, TRST, TCLK, TDI, and TDO. Note that TRST pin may not be present in some embodiments. The TMS, TRST, and TCLK pins are coupled to a JTAG state machine **210**. JTAG state machine **210** is a state machine providing output signals SHFTIR, CLKIR, UPDTIR, SHFTDR, CLKDR, and UPDTDR based on the TMS, TRST, TCLK inputs. JTAG state machine **210** controls the sequential operation of the circuitry.

TDI is a serial input to an instruction register **220**, and TDO is a serial output. JTAG state machine **210** controls the serially shifting of an instruction from TDI into instruction register **220**. The instruction may be serially shifted out through TDO. Further, the instruction may also be output in parallel via INST**0**, INST**1**, and INST**2** lines. In the JTAG circuitry, there are also data registers (such as a boundary-scan register) that are not explicitly shown in FIG. 2. A description of the JTAG data registers may be found in Application Note 39.

The instruction is coupled to a JTAG boundary control scan control logic **230** via parallel INST**0**, INST**1**, and INST**2** lines. JTAG boundary scan control logic **230** generates JTAG scan control signal **235**. JTAG scan control signals **235** are routed to the appropriate JTAG circuitry to control JTAG operation. Further, JTAG boundary scan control logic **230** also generates a mode select signal **240** provided to a programming mode decoder **250**. Programming mode decoder also has inputs from mode select pin **1** (**252**) and mode select pin **2** (**253**), and generates programming mode signals **255**.

FIG. **3** shows a specific implementation of circuitry within JTAG boundary scan control logic **230**. The input signals are INST**0**, INST2, NINST**0**, NINST1, and NINST2. NINST**0**, NINST1, and NINST2 are the complements of INST**0**, INST1, and INST2, respectively. For example, NINST**0**, NINST1, and NINST2 may be obtained by inverting the INST**0**, INST1, and INST2 using inverters. Output signals are JTEST, SAMPLE, and EXTEXT.

The circuitry includes NAND gates **310**, **320**, and **330**. NAND gate **310** has as inputs NINST**0**, NINST1, and INST2. NAND **310** couples to an inverter **340** to output JTEST. NAND gate **320** has as inputs INST**0**, NINST1, and INST2. NAND **320** couples to an inverter **350** to output SAMPLE. NAND gate **330** has as inputs NINST**0**, NINST1, and NINST2. NAND **330** couples to an inverter **360** to output EXTEST.

The JTAG boundary scan control logic **230** circuitry determines which mode the PLD will be in based on the instruction input by the user. In this embodiment, the instruction has three bits, INST**0**, INST1, and INST2. In other embodiments, there may be more or less than three bits depending on the desired number of different instructions. For example, in some embodiments, the instruction has ten bits. With three bits, up to eight different instructions can be implemented. With ten bits, up to 2^{10} different instructions can be implemented.

The circuitry in FIG. **3** decodes the instructions as follows. A "001" indicates a JTEST mode (where INST**0** is 0, INST1 is 0, and INST2 is 1). A "101" indicates a SAMPLE mode. A "001" indicates an EXTEST mode. SAMPLE and EXTEST are JTAG modes. JTEST is a configuration or programming mode. Therefore, by using the same dedicated pins are used to input JTAG instructions, a programming mode may also be indicated. This means a separate dedicated pin to indicate a programming mode is not needed, thus saving a dedicate pin which may be used instead for user I/O. Furthermore, there may be other JTAG modes (not shown in FIG. **3**) such as BYPASS which is typically indicated by a "111" instruction.

In the SAMPLE and EXTEXT modes, the corresponding SAMPLE and EXTEST signals will be logic high. And, in the JTEST mode, the JTEST signal will be a logic high. SAMPLE and EXTEXT are examples of JTAG control signals **235**. JTEST is an example of mode select signal **240**.

FIG. **4** shows circuitry for programming mode decoder **250**. Inputs are ENA, JTEST, and MSEL. ENA is an enable signal to enable decoder **250**. JTEST is generated by JTAG boundary scan control logic **230** (such as the circuitry shown in FIG. **3**). MSEL is representative of mode select a pins **252** and **253**. However, the implementation FIG. **4** only shows one mode select pin in order to illustrate the principles of the present invention. In practice, there may be as many or as few mode select pins as desired to obtain the number of desired modes.

Outputs of decoder **250** are TEST, SCAN, ASYNC-SERIAL, and SERIAL. These signals are routed to the appropriate programming circuitry to configure the PLD. The programming circuitry may generate high voltages such

as those used in the programming of Flash, EEPROM, EPROM, and other nonvolatile memory cells. The programming circuitry may also be used to configure other types of memory cells such as SRAM and DRAM cells.

The circuitry includes NAND gates **410**, **420**, **430**, and **440**. NAND gate **410** has inputs JTEST, ENA, and MSEL. NAND gate **410** outputs through a pair of serially coupled inverters to generate TEST. NAND gate **420** has inputs JTEST, ENA, and BB (i.e., complement of MSEL). NAND gate **420** outputs through a pair of serially coupled inverters to generate SCAN. NAND gate **430** has inputs CB (i.e., complement of JTEST), ENA, and MSEL. NAND gate **430** outputs through a pair of serially coupled inverters to generate ASYNC-SERIAL. NAND gate **440** has inputs CB, ENA, and BB. NAND gate **440** outputs through a pair of serially coupled inverters to generate SERIAL.

In operation, a "test" programming mode is entered when ENA is high, JTEST is high, and MSEL is high. A "scan" programming mode is entered when ENA is high, JTEST is high, and MSEL is low. An "async-serial" programming mode is entered when ENA is high, JTEST is low, and MSEL is high. A "serial" mode is entered when ENA is high, JTEST is low, and MSEL is low.

In practice, there are many implementations of the boundary scan control logic circuitry **230** and programming mode decoder **250** shown in FIGS. **3** and **4**. Other implementations may use other selections for the decoding and other logical structures including AND and OR gates or look-up tables, to name a few examples.

For example, other specific instructions may be used to indicate a programming mode. Standard JTAG instructions are identified by 101, 000, and 111. Consequently, a programming mode control signal may be implemented by using an instruction not already used by JTAG. The available instructions are 001, 010, 011, 100, and 110. In FIG. **3**, the choice of the specific instruction to indicate the JTEST programming mode was 001. However, any of the other available instructions could have been selected, and the appropriate changes made to the circuitry.

Furthermore, FIG. **3** only shows a single JTEST programming mode; however, circuitry may provide for more than one programming mode signal. With a 3-bit instruction, there can be up to five programming mode instructions. The circuitry can be modified to provide more than one programming mode instruction, and thus save greater numbers of dedicated mode select pins.

The circuitry shown in FIGS. **3** and **4** illustrates (by way of an specific example) a technique to eliminate one mode select pin by implementing a JTEST instruction. The JTEST instruction is recognized by the JTAG circuitry to indicate a programming mode. Using the JTEST signal, programming mode decoder **250** provides four modes, taking as input only one dedicated mode select pin. Without the JTEST instruction, two dedicated mode select pins would have been required to have four different programming modes. The JTEST instruction saves one mode select pin. Therefore, using the technique of the present invention, fewer dedicated pins are required to implement the programming modes, leaving more external pins for user I/O.

In further embodiments of the present invention, dedicated pins to indicate the programming modes may be eliminated altogether. In that case, the programming modes would be determined entirely by the instruction in instruction register **220**. And there may be multiple JTEST signals. For example, an instruction may be decoded to provide JTEST1, JTEST2, and JTEST3 signals used to distinguish between up to eight programming modes. As discussed above, the number of available programming modes depends on the number of available instructions not being used to implement JTAG modes.

The techniques and circuitry of the present invention are also applicable for in-system programming (ISP) of a PLD, where the PLD is programmed while resident on a printed circuit board.

FIG. **5** shows a flow diagram of a technique of the present invention. The technique of the present invention permits the programming or configuration of an integrated circuit using the JTAG circuitry. In a step **501**, an instruction is input into JTAG instruction register **220** of the integrated circuit. The instruction may be serially shifted in via the TDI pin according to the control signals from JTAG state machine **210**. In a specific embodiment, the instruction may have three bits INST0, INST1, and INST2.

In a step **505**, the instruction in the instruction register is decoded. The instruction is passed in parallel to JTAG boundary scan control logic **220**. JTAG boundary scan control logic **220** generates the appropriate control signal to indicate a JTAG mode or a programming mode. For example, SAMPLE and EXTEST are JTAG modes, and JTEST is a programming mode.

In a step **510**, the JTEST signal is generated by JTAG boundary scan control logic **220** to indicate a programming mode. The JTEST signal may be implemented using an available instruction which is not used as a JTAG instruction.

In a step **515**, the JTEST signal is passed to programming mode decoder **250**. In a step **520**, using the JTEST signal, the programming mode decoder **250** generates programming mode signals **255** (such as TEST, SCAN, ASYNC-SERIAL, and SERIAL) that are passed to the programming circuitry.

Based on programming mode signals **255**, the integrated circuit will be configured by the programming circuitry. The configuration of the integrated circuit may be in an in-system programming (ISP) mode.

The present invention may be used in conjunction and is compatible with other techniques to effectively increase the available number of user I/O pins, such as described in U.S. patent application Ser. No. 09/094,226, filed Jun. 9, 1998, now U.S. Pat. No. 6,314,550, which is incorporated by reference.

Another technique to increase the number of pins is to permit the use of the JTAG pins for user I/O when JTAG is not used by the user. To implement the JTAG standard in an integrated circuit, the integrated circuit needs at least four pins: TCLK, TMS, TDIN, and TDO. These are dedicated pins for accessing JTAG functionality. However, for customers who do not use JTAG, these pins are not used. The technique of the present invention allows these customers to use the JTAG pins as regular I/O pins. The technique of the present invention is especially useful for programmable logic devices (PLDs), field programmable gate arrays (FPGAs), and many other types of integrated circuits.

In the method of the present invention, the information whether JTAG operation is enabled or disabled is encoded in an option register bit. After power up of the integrated circuit, the default state of option register allows these four pins to be used as JTAG pins. Thus, JTAG operation is enabled. After the option register bit is programmed, there are two cases.

(1) The customer may choose to use JTAG, and the option register is configured to reflect this. Then, these four pins will continue to function as JTAG pins.

(2) In the case the customer chooses not to use JTAG, the option register is configured to reflect this. The four JTAG pins will be disconnected from the JTAG circuitry. JTAG operation will not be enabled. After the device enters the user mode, these four pins can be used as regular I/O pins, thus avoiding the waste of these pins when JTAG is not used.

The configuration of the option register may be held using memory cells such as SRAM, EPROM, EEPROM, Flash, RAM, and many others. The configuration information may be nonvolatile.

During programming, the JTAG state machine stays in the reset state regardless the state of JTAG pins.

An advantage of the method of the present invention is to allow four more I/O pins for those customers who do not use JTAG. These customers can treat the four pins as regular I/O pins during both programming and user mode. Further, there is no "difficult to use" problem.

FIGS. **6**, **7**, and **8** show a circuit implementation for an integrated circuit with configurably or selectably enabled and disabled JTAG pins.

FIG. **6** shows circuitry which may be used for the TDI, TCLK, and TMS input pins. Pin **610** is the I/O pin of the integrated circuit, and is coupled to an output buffer **615**. Output buffer **615** has transistor drivers coupled to a noisy positive supply VCCN and noisy ground supply VSSN. VCCN and VSSN are distinguished from quiet positive and ground supplies VCCQ and VSSQ, respectively. Some degree of isolation is achieved by separating the quiet and noisy supplies. However, in some implementations, there may be only VCC and VSS supply pins, where there are not separate noisy and quiet supplies.

Output buffer **615** is a data output buffer for drive data to pin **610**. Output data is input at a DIN input. An OEB input controls whether pin **610** is tristated. A RNSLEW input controls whether the slew rate at the drivers of the output buffer are slowed in order to minimize or prevent ground or power bounce. A DATX output passes data from pin **610** to an input buffer for the integrated circuit. The input buffer includes inverters INV7 and INV6. An output of INV6 drives the internal circuitry.

An RJTAG input to the circuitry determines whether JTAG functionality is enabled or disabled. A JTAG input buffer is represented by block, the details of which are shown in FIG. **7**. The JTAG input buffer includes inverters **710** and **720** and a transmission gate **725**. An input of inverter **710** is coupled to DATX (see FIG. **6**). A control electrode of an NMOS transistor of transmission gate **725** is coupled to RJTAG. A control electrode of a PMOS transistor of transmission gate **725** is coupled through an inverter **725** to RJTAG. An output of inverter **726** is also coupled to a gate of an NMOS transistor **740**, which is coupled between an output **628** of the JTAG input buffer and ground.

When RJTAG is a logic high, the JTAG input buffer **620** is enabled to pass data to the internal JTAG circuitry and JTAG functionality is permitted.

When RJTAG is a logic low, JTAG functionality if disabled. This is occurs by disabling the JTAG input buffer. Transmission gate **725** is turned off, decoupling an output of inverter **720** from output **628** of the JTAG input buffer. Transistor **740** is turned on in order to ground output **628**. A PMOS transistor **635** is turned on in order to couple VCC to the input of the JTAG input buffer **620**, ensuring the input is a logic high. Then, inverters **720** and **710** will be in a known state, and not consume unnecessary power. RJTAG will control the output buffer **615** to function as a user I/O pin.

FIG. **8** shows circuitry which may be used for a TDO output pin. Depending on the states of RJTAG and JOEB, the JTAG functionality may be disabled. The input buffer (i.e., INV6 and INV7) and output buffer **615** are as described above. A circuit block is a JTAG output buffer **810** for outputting JTAG data. This JTAG data is input to the JTAG output buffer through the JDIN pin. An output of the JTAG output buffer **810** is coupled to pin **610**.

When RJTAG is logic high and JOEB is logic low, JTAG functionality will be enabled. When RJTAG is logic low and JOEB is logic high, JTAG functionality will be disabled. Specifically, the JTAG output buffer will be tristated, and output buffer **615** will function similarly as for a user I/O pin.

The foregoing description of preferred embodiments of the invention has been presented for the purposes of illustration and description. It is not intended to be exhaustive or to limit the invention to the precise form described, and many modifications and variations are possible in light of the teaching above. The embodiments were chosen and described in order to best explain the principles of the invention and its practical applications to thereby enable others skilled in the art to best utilize and practice the invention in various embodiments and with various modifications as are suited to the particular use contemplated. It is intended that the scope of the invention be defined by the following claims.

What is claimed is:

1. A method of configuring a programmable integrated circuit comprising:

providing an instruction to a JTAG instruction register;

passing the instruction to a JTAG boundary scan control logic block wherein the JTAG boundary scan control logic generates JTAG scan control signals;

generating in the JTAG boundary scan control logic block a control signal;

passing the control signal to a programming mode decoder; and

based on the control signal, generating a programming mode signal to place the programmable integrated circuit in a configuration mode.

2. The method of claim **1** wherein the instruction is not an IEEE 1149.1 JTAG instruction.

3. The method of claim **1** wherein the instruction is shifted serially into the JTAG instruction register.

4. The method of claim **1** wherein the instruction is passed in parallel to the JTAG boundary scan control logic block.

5. The method of claim **1** wherein the control signal is not a JTAG scan control signal.

6. The method of claim **1** further comprising:

in the configuration mode, loading configuration information into the programmable integrated circuit through non-JTAG external pins.

7. The method of claim **1** further comprising:

configuring programmable memory cells of the programmable integrated circuit in the configuration mode.

8. The method of claim **7** wherein the programmable memory cells are EEPROM or Flash cells.

9. The method of claim **7** wherein the programmable memory cells are SRAM cells.

10. The method of claim **1** wherein the programming mode decoder comprises:

a plurality of logic gates, each having an input coupled to the control signal and at least two inputs coupled to mode select signals, and providing four separate programming mode signals.

11. A method of configuring a programmable integrated circuit comprising:

storing in a JTAG instruction register an instruction indicating the programmable integrated circuit is to be configured;

placing the programmable integrated circuit in a configuration mode based on the instruction stored in the JTAG instruction register; and

configuring programmable memory cells of the programmable integrated circuit in the configuration mode.

12. The method of claim **11** further comprising:

decoding the instruction using a JTAG boundary scan control logic block; and

determining the instruction is not a JTAG scan control signal.

13. The method of claim **11** wherein configuring the programmable integrated circuit comprises imprinting the programmable integrated circuit with a desired pattern.

14. The method of claim **11** further comprising:

in the configuration mode, selecting a configuration mode operation based on the instruction stored in the JTAG instruction register and a mode select pin.

15. The method of claim **11** wherein the programmable memory cells are EEPROM or Flash cells.

16. The method of claim **11** wherein the programmable memory cells are SRAM cells.

17. A method of operating a programmable logic device comprising:

serially inputting an instruction to an instruction register of the programmable logic device;

decoding the instruction in a control logic block;

when the instruction is an IEEE 1149.1 JTAG instruction, performing IEEE 1149.1 JTAG operations in the programmable logic device;

when the instruction is a configuration instruction, generating a control signal in the control logic block;

in response to the control signal, generating a programming mode signal to place the programmable logic device into a configuration mode.

18. The method of claim **17** further comprising:

when in the configuration mode, permitting a user to configure the programmable logic device.

19. The method of claim **17** wherein the configuration instruction is not a IEEE 1149.1 standard JTAG instruction.

20. The method of claim **17** further comprising:

when the instruction is not the configuration instruction, not placing the programmable logic device in the configuration mode.

21. The method of claim **17** further comprising:

when in the configuration mode, permitting a user to select from a plurality of different modes within the configuration mode.

22. The method of claim **21** wherein the user's selection of one of the plurality of different modes is input by way of parallel instruction bits provided to the programmable logic device.

23. A technique of programming a programmable logic device while it is resident on a system board comprising the method recited in claim **17**.

24. The method of claim **17** wherein the JTAG instruction comprises a plurality of bits clocked into the programmable logic device.

25. The method of claim **17** wherein the control logic block comprises a plurality of NAND gates, each coupled to the instruction register.

26. The method of claim 17 further comprising:

configuring programmable memory cells of the programmable integrated circuit in the configuration mode.

27. The method of claim 26 wherein the programmable memory cells are EEPROM or Flash cells.

28. A method of placing a programmable logic integrated circuit into its configuration mode to program memory cells of the programmable logic integrated circuit comprising inputting a configuration instruction into a register of the programmable logic integrated circuit, wherein the register is also used for IEEE 1149.1 JTAG operations when IEEE 1149.1 JTAG instructions are input into the register.

29. The method of claim 28 wherein the memory cells are EEPROM or Flash cells.

* * * * *

Exhibit B

US005563592A

United States Patent [19]

Cliff et al.

[11] Patent Number: 5,563,592

[45] Date of Patent: Oct. 8, 1996

[54] PROGRAMMABLE LOGIC DEVICE HAVING A COMPRESSED CONFIGURATION FILE AND ASSOCIATED DECOMPRESSION

[75] Inventors: **Richard G. Cliff**, Milpitas; **L. Todd Cope**, San Jose, both of Calif.

[73] Assignee: **Altera Corporation**, San Josc, Calif.

[21] Appl. No.: **156,561**

[22] Filed: **Nov. 22, 1993**

[51] **Int. Cl.**^{6} **H03M 7/46**

[52] **U.S. Cl.** **341/63**; 364/715.02

[58] **Field of Search** 341/63, 64; 364/715.03, 364/715.02, 715.09

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,446,516	5/1984	Nishimura	341/63
4,791,660	12/1988	Oye et al.	379/88
5,258,932	11/1993	Matsuzaki	364/578
5,260,610	11/1993	Pedersen et al.	326/41
5,260,611	11/1993	Cliff et al.	326/39
5,440,718	8/1995	Kumagai et al.	395/481

FOREIGN PATENT DOCUMENTS

0500267 8/1992 European Pat. Off. .

OTHER PUBLICATIONS

Jock Tomlinson et al., "Designing with reprogrammable FPGAs", *Australian Electronics Engineering*, Feb. 1993, 67–70.

Shih–Fu Chang et al., "Designing High–Throughput VLC Decoder Part I—Concurrent VLSI Architectures", *IEEE Transactions on Circuits and Systems for Video Technology*, (Jun. 1992) 2:2:187–96.

Ming–Ting Sun et al., "High–Speed Programmable ICs for Decoding of Variable–Length Codes", *SPIE*, (1989) 1153:28–39.

Shih–Fu Chang et al., "VLSI Designs for High–Speed Huffman Decoder", *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, (1991), pp. 500–503.

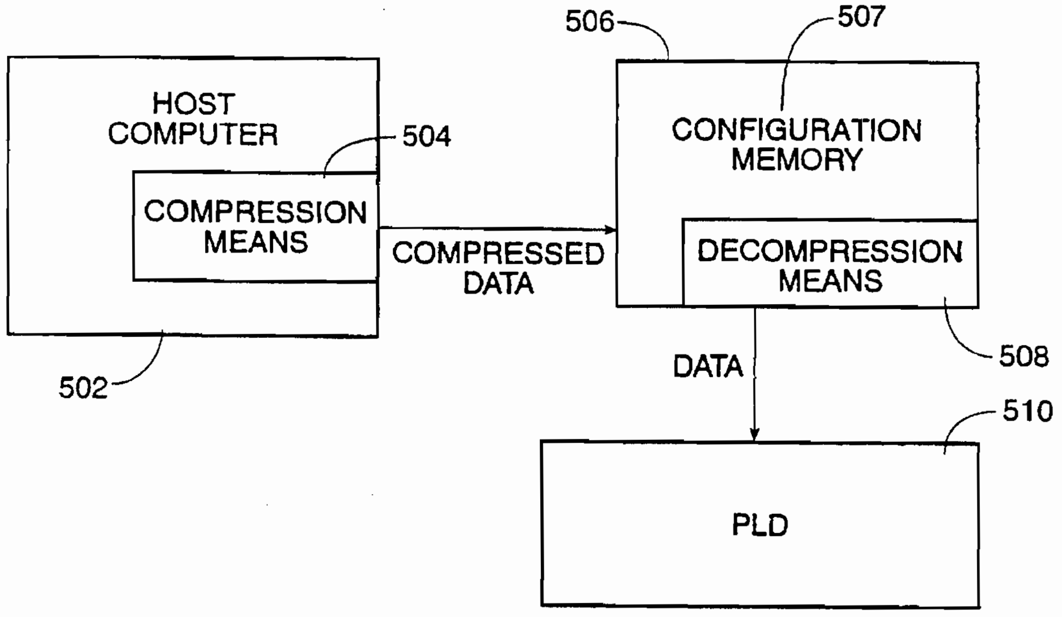
J. C. Vermeulen, "Logarithmic Counter Array with Fast Access", *IBM Technical Disclosure Bulletin*, (Nov. 1984), 27:6:3380–3381.

Primary Examiner—Howard L. Williams
Attorney, Agent, or Firm—Townsend and Townsend and Crew LLP

[57] **ABSTRACT**

A method of utilizing compression in programming programmable logic devices is disclosed. The present invention compresses the configuration file to be used in programming a programmable logic device. The compression step reduces the size of the configuration file. The reduction in the size of the compression file results in the reduction in the size of the memory device used to store the configuration file before it is used to program the programmable logic device.

34 Claims, 8 Drawing Sheets



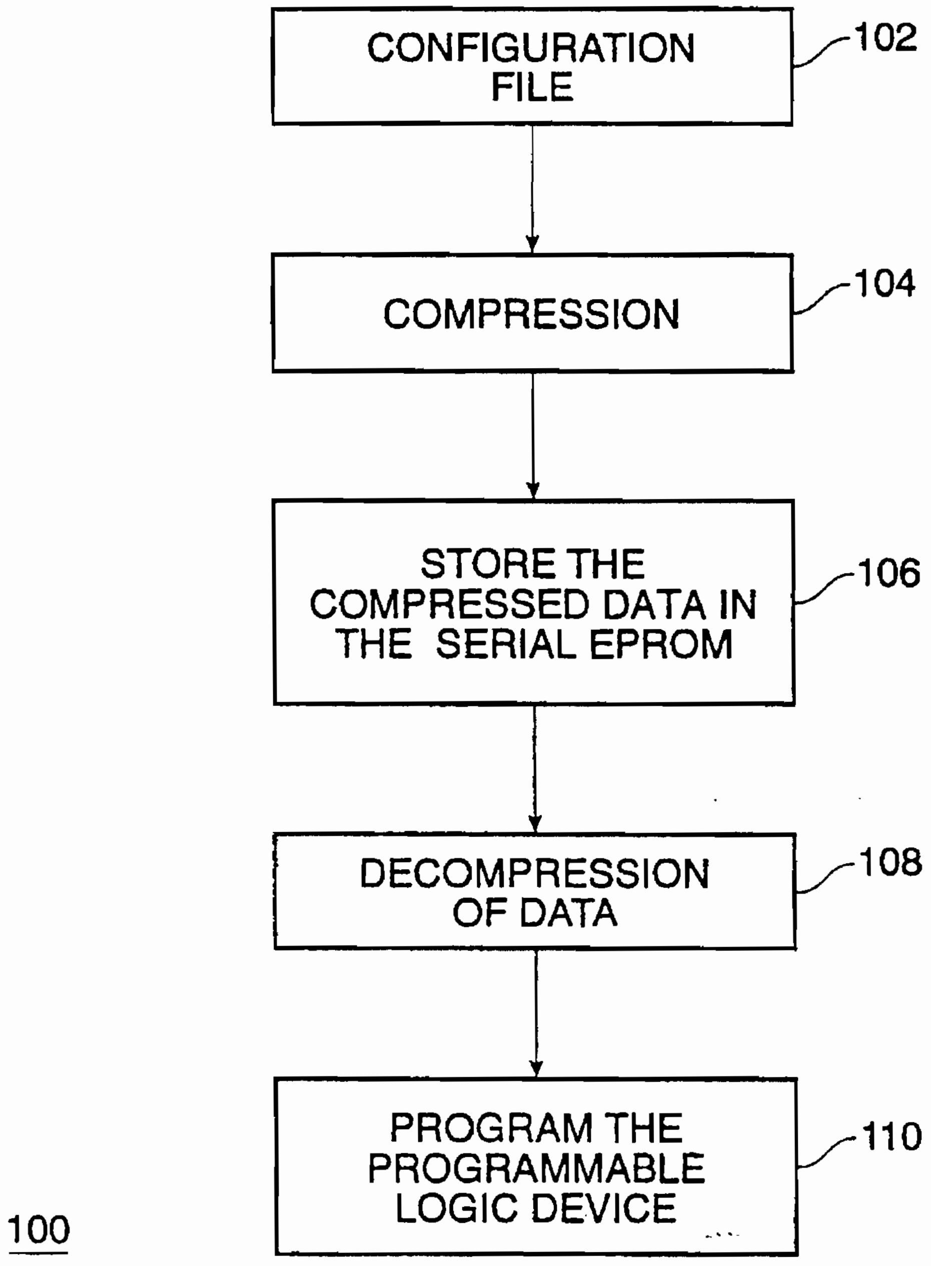

CONFIGURATION FILE
102
COMPRESSION
104
STORE THE COMPRESSED DATA IN THE SERIAL EPROM
106
DECOMPRESSION OF DATA
108
PROGRAM THE PROGRAMMABLE LOGIC DEVICE
110
100

FIG. 1

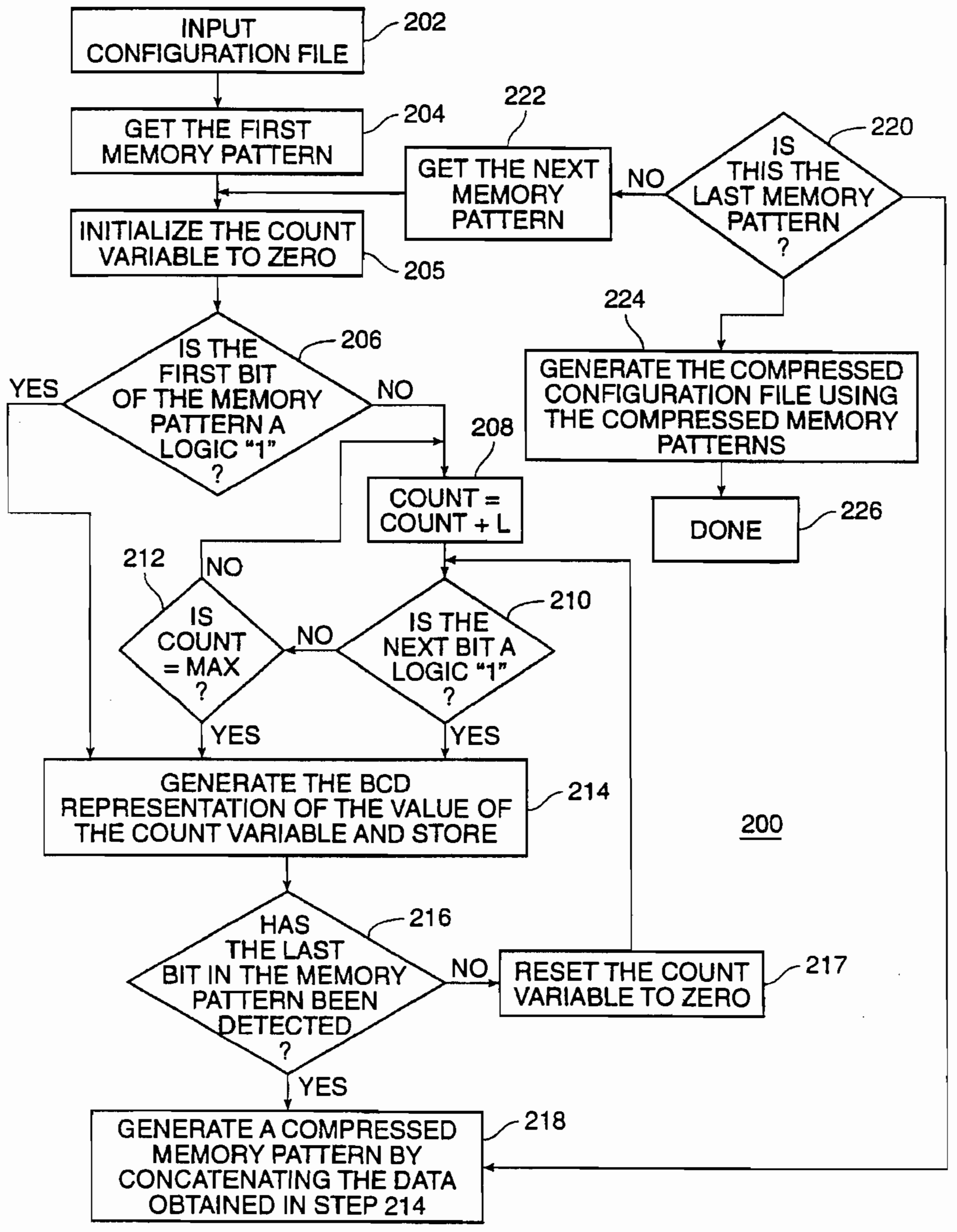

INPUT CONFIGURATION FILE
202
GET THE FIRST MEMORY PATTERN
204
222
GET THE NEXT MEMORY PATTERN
NO
IS THIS THE LAST MEMORY PATTERN ?
220
INITIALIZE THE COUNT VARIABLE TO ZERO
205
224
GENERATE THE COMPRESSED CONFIGURATION FILE USING THE COMPRESSED MEMORY PATTERNS
IS THE FIRST BIT OF THE MEMORY PATTERN A LOGIC "1" ?
206
YES
NO
208
COUNT = COUNT + L
DONE
226
212
NO
IS COUNT = MAX ?
NO
IS THE NEXT BIT A LOGIC "1" ?
210
YES
YES
GENERATE THE BCD REPRESENTATION OF THE VALUE OF THE COUNT VARIABLE AND STORE
214
200
HAS THE LAST BIT IN THE MEMORY PATTERN BEEN DETECTED ?
216
NO
RESET THE COUNT VARIABLE TO ZERO
217
YES
GENERATE A COMPRESSED MEMORY PATTERN BY CONCATENATING THE DATA OBTAINED IN STEP 214
218

FIG. 2

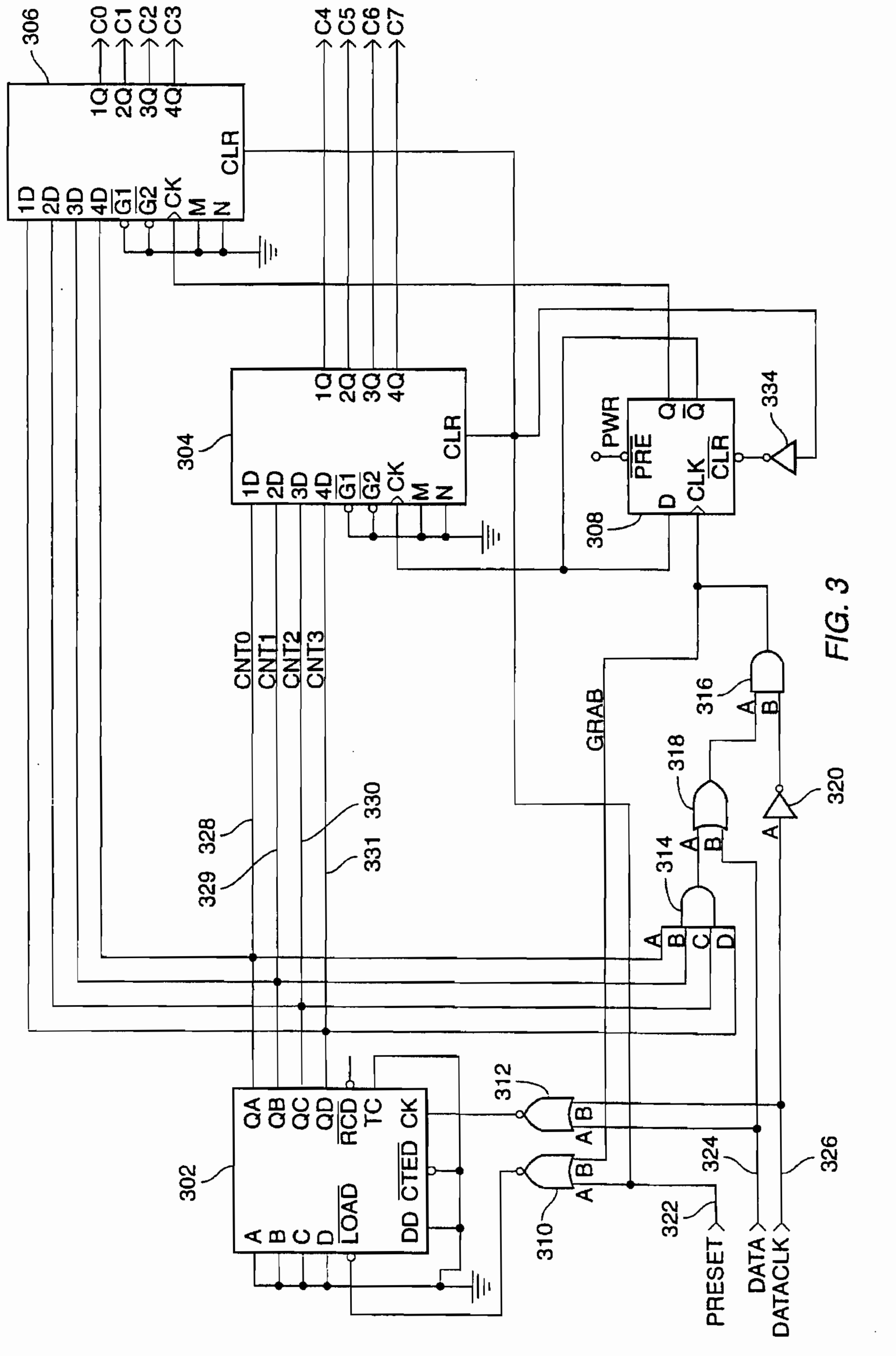

306
C0
C1
C2
C3
C4
C5
C6
C7
1D 2D 3D 4D G1 G2 CK M N
1Q 2Q 3Q 4Q
CLR
304
308
334
PWR
PRE
CLR
CLK
D
Q
Q̄
CNT0
CNT1
CNT2
CNT3
GRAB
316
318
320
314
328
329
330
331
A B C D
302
QA QB QC QD RCO TC CK
A B C D LOAD DD CTED
312
310
322
324
326
PRESET
DATA
DATACLK

FIG. 3

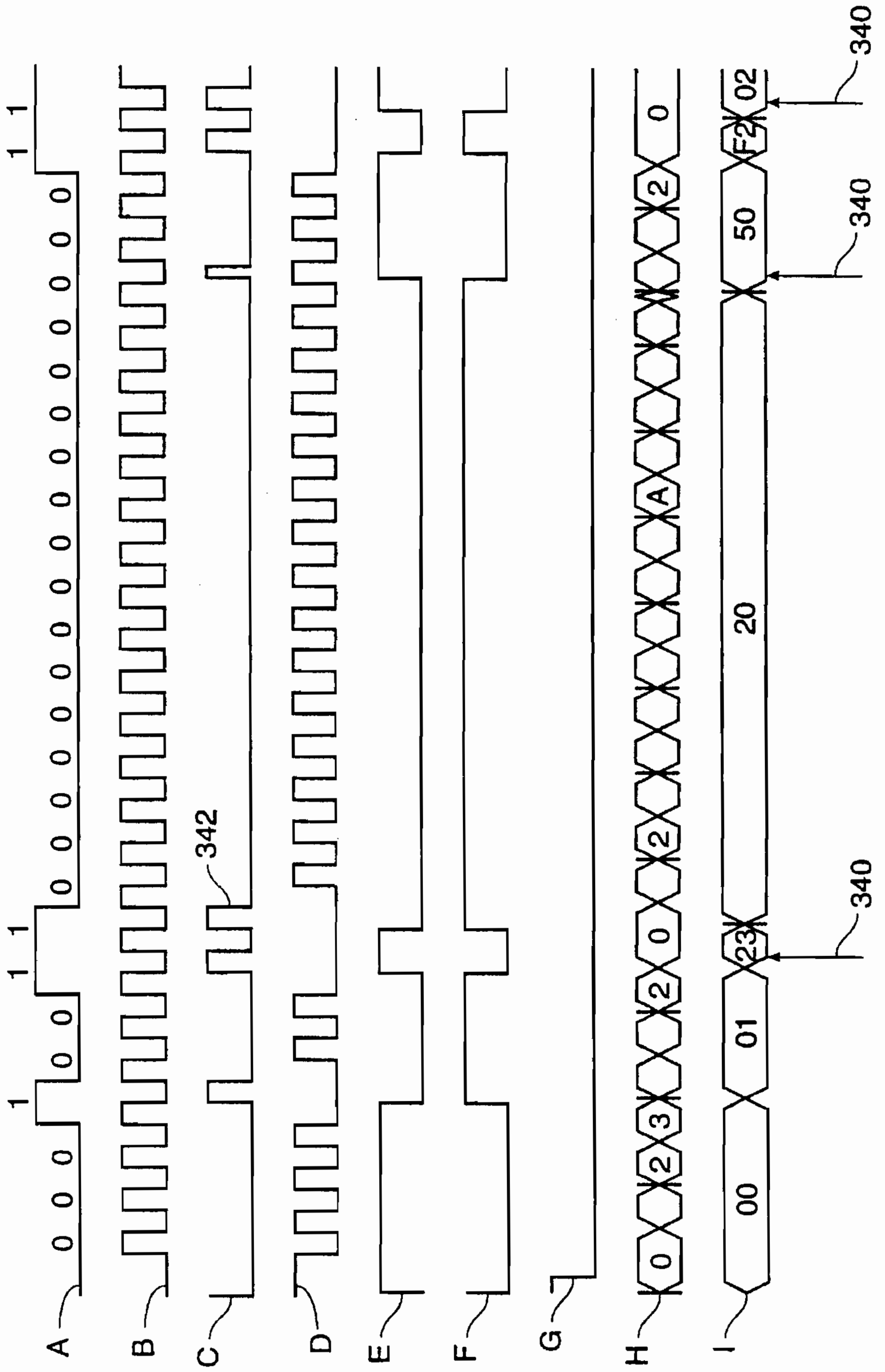

A
B
C
D
E
F
G
H
I
0 0 0 1 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
342
340
340
340
0 2 3 2 0 2 A 2 0
00 01 23 20 50 F2 02

FIG. 4

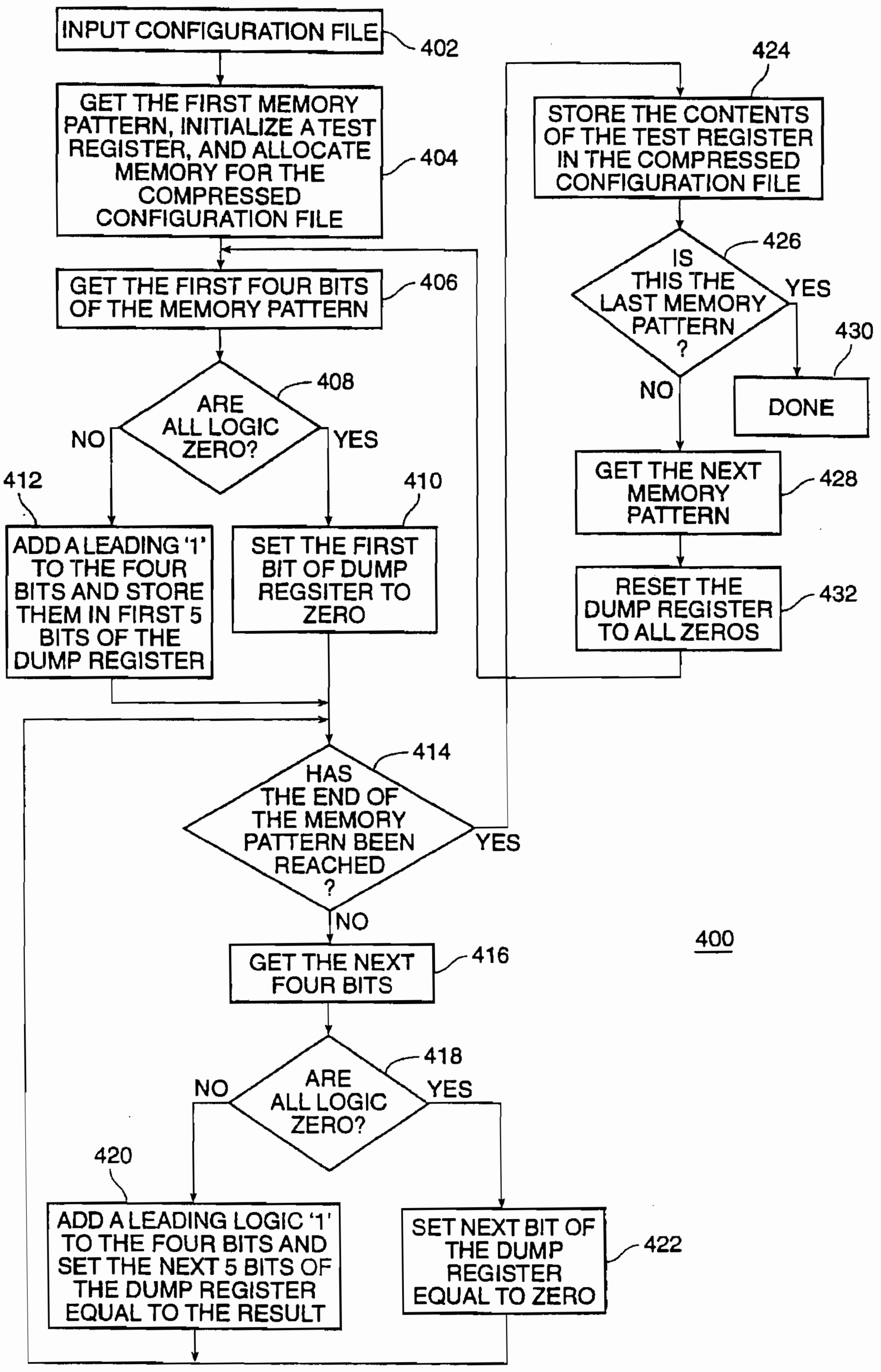

INPUT CONFIGURATION FILE
402
GET THE FIRST MEMORY PATTERN, INITIALIZE A TEST REGISTER, AND ALLOCATE MEMORY FOR THE COMPRESSED CONFIGURATION FILE
404
GET THE FIRST FOUR BITS OF THE MEMORY PATTERN
406
ARE ALL LOGIC ZERO?
408
NO
YES
412
ADD A LEADING '1' TO THE FOUR BITS AND STORE THEM IN FIRST 5 BITS OF THE DUMP REGISTER
410
SET THE FIRST BIT OF DUMP REGSITER TO ZERO
HAS THE END OF THE MEMORY PATTERN BEEN REACHED ?
414
YES
NO
GET THE NEXT FOUR BITS
416
ARE ALL LOGIC ZERO?
418
NO
YES
420
ADD A LEADING LOGIC '1' TO THE FOUR BITS AND SET THE NEXT 5 BITS OF THE DUMP REGISTER EQUAL TO THE RESULT
SET NEXT BIT OF THE DUMP REGISTER EQUAL TO ZERO
422
424
STORE THE CONTENTS OF THE TEST REGISTER IN THE COMPRESSED CONFIGURATION FILE
IS THIS THE LAST MEMORY PATTERN ?
426
YES
430
DONE
NO
GET THE NEXT MEMORY PATTERN
428
RESET THE DUMP REGISTER TO ALL ZEROS
432
400

FIG. 5

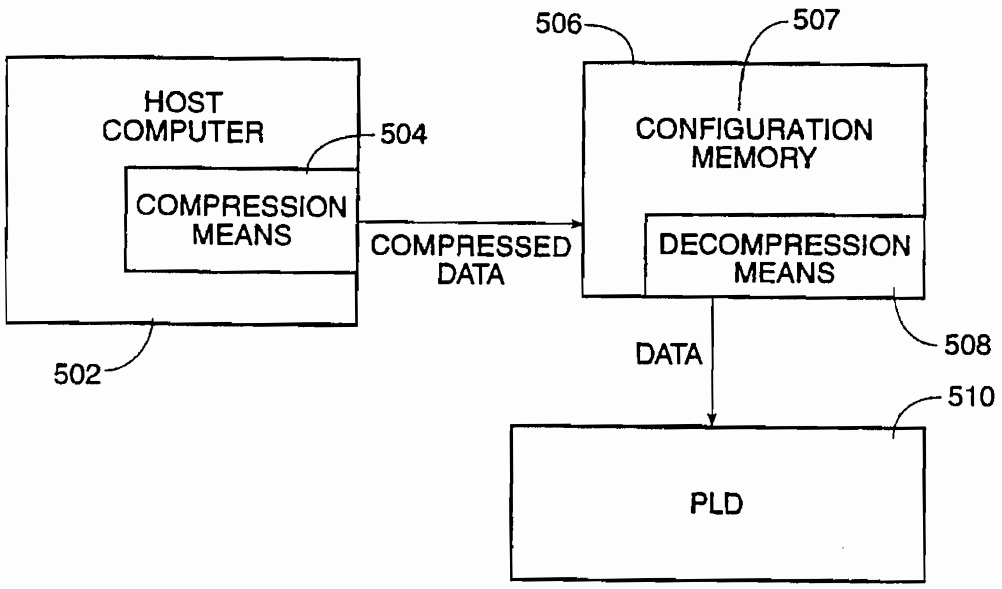

HOST COMPUTER
COMPRESSION MEANS
504
502
COMPRESSED DATA
506
507
CONFIGURATION MEMORY
DECOMPRESSION MEANS
508
DATA
510
PLD

FIG. 6

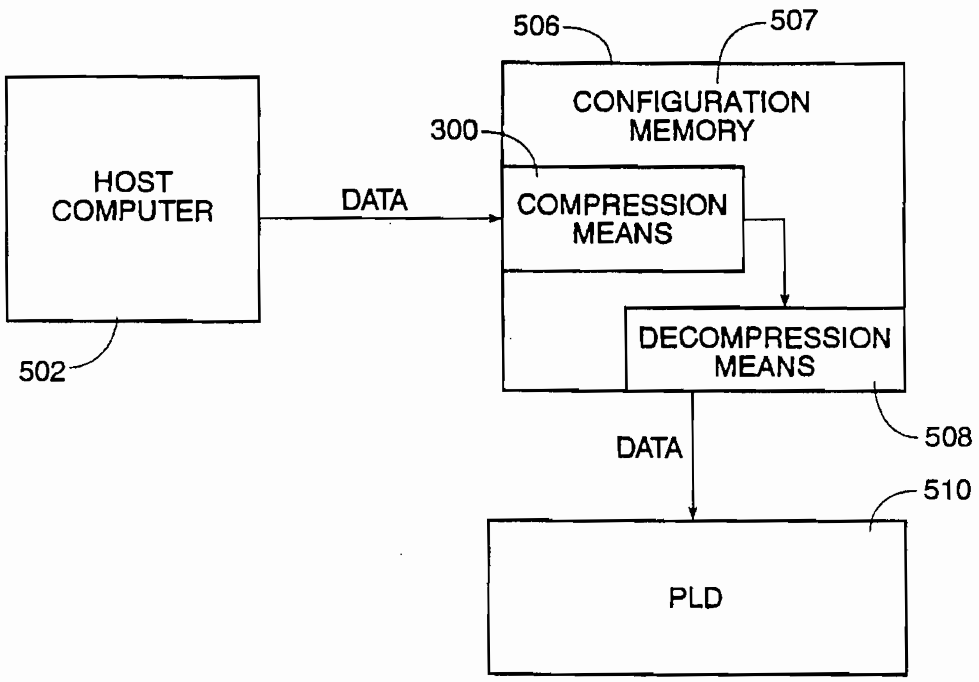

HOST COMPUTER
502
DATA
506
507
CONFIGURATION MEMORY
300
COMPRESSION MEANS
DECOMPRESSION MEANS
508
DATA
510
PLD

FIG. 7

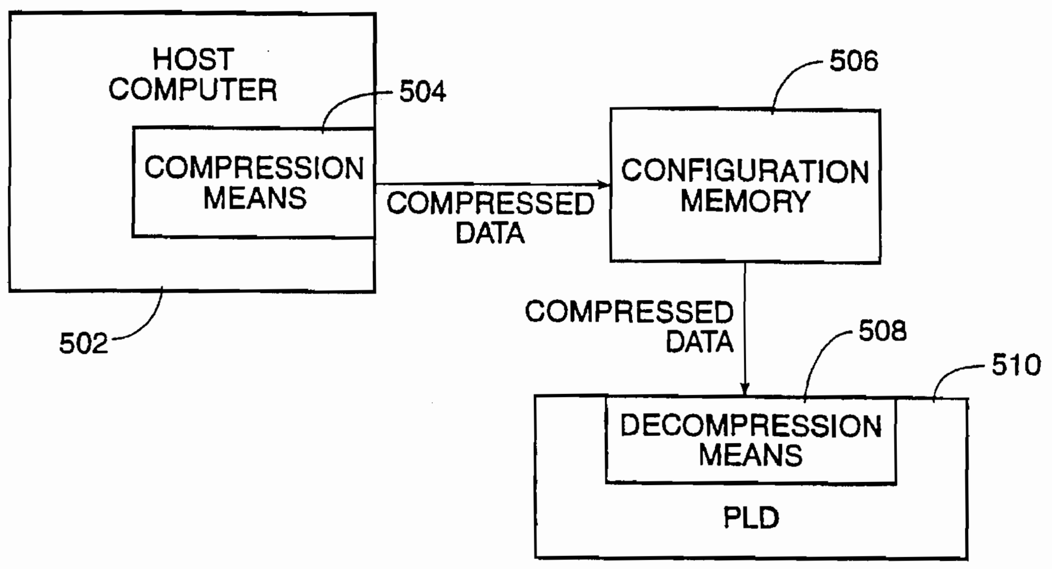

HOST COMPUTER
504
COMPRESSION MEANS
COMPRESSED DATA
506
CONFIGURATION MEMORY
502
COMPRESSED DATA
508
510
DECOMPRESSION MEANS
PLD

FIG. 8

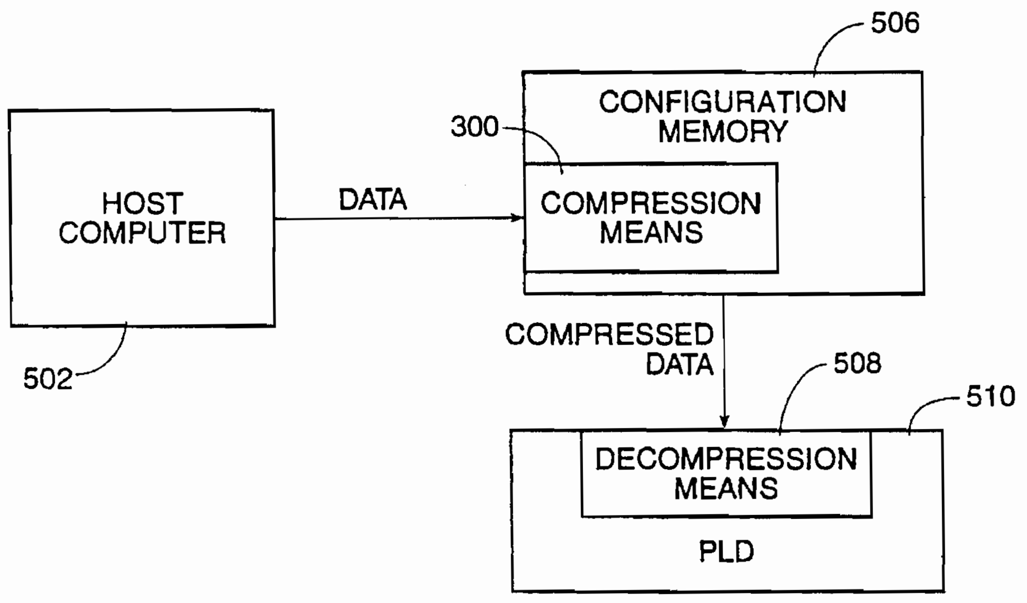

506
CONFIGURATION MEMORY
300
HOST COMPUTER
DATA
COMPRESSION MEANS
502
COMPRESSED DATA
508
510
DECOMPRESSION MEANS
PLD

FIG. 9

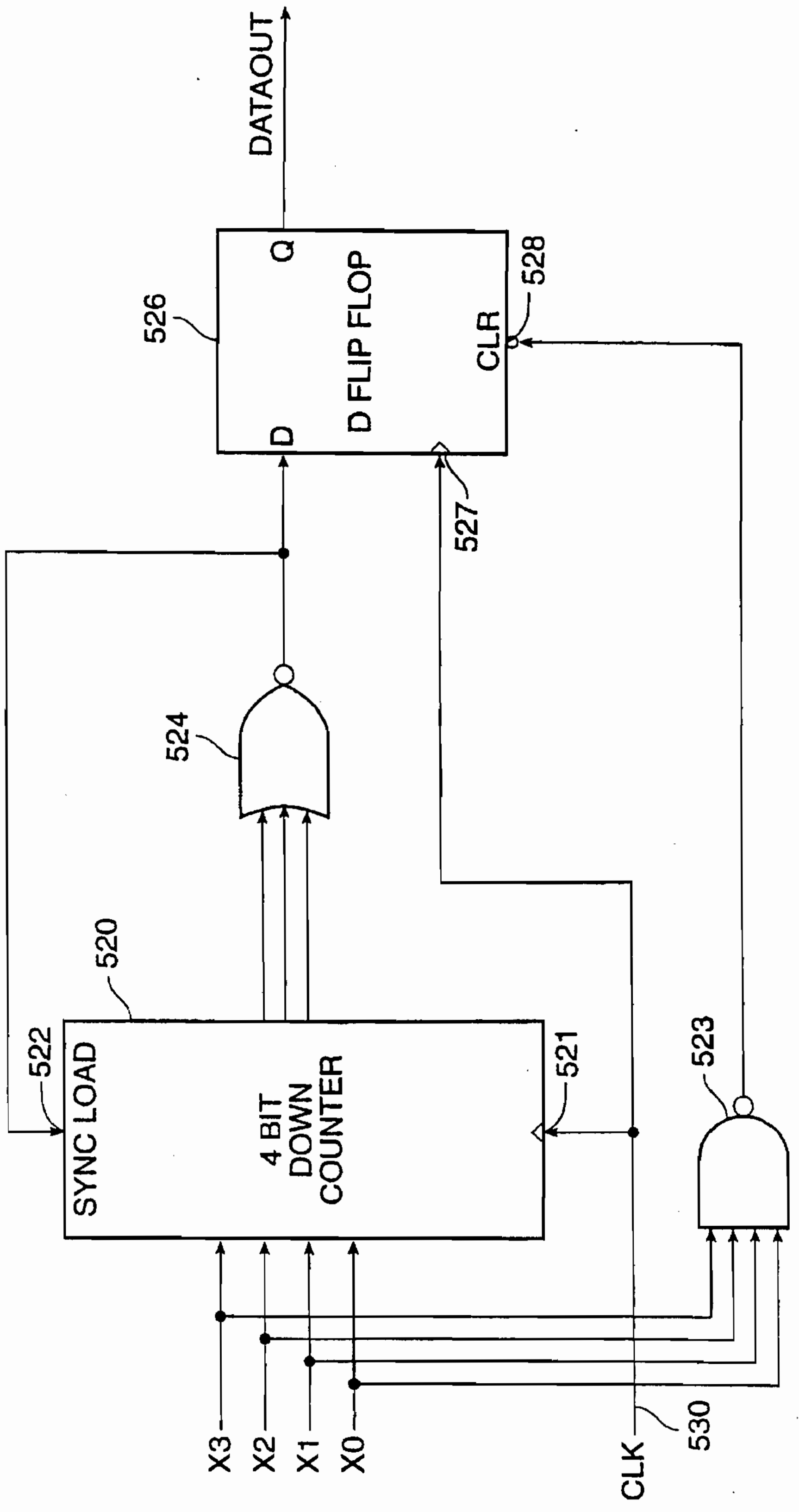

DATAOUT
Q
D FLIP FLOP
D
CLR
526
528
527
524
520
522
SYNC LOAD
4 BIT DOWN COUNTER
521
523
X3
X2
X1
X0
CLK
530

FIG. 10

508

PROGRAMMABLE LOGIC DEVICE HAVING A COMPRESSED CONFIGURATION FILE AND ASSOCIATED DECOMPRESSION

BACKGROUND OF THE INVENTION

The present invention relates to the field of data processing. In particular, it relates to the use of data compression in for transmitting and storing configuration data for programming programmable logic devices.

Today's advanced technology has provided for design, development and manufacturing of complicated programmable logic devices. Such devices include those described in U.S. Pat. Nos. 5,260,610 and 5,260,611, incorporated herein by reference for all purposes. These devices include numerous programmable elements to provide flexibility. By programming these programmable elements, the user defines the function that the programmable logic device must perform. Configuration files are used to program the programmable logic devices. A typical configuration file includes at least one memory pattern. The memory pattern includes a series of low "0" and high "1" bits which are used to program the individual programming elements. Configuration files are stored on the PLD or in memory associated with the PLD and are loaded onto the PLD at power-up or when the PLD system receives a signal to reconfigure the PLD.

Typically, a configuration file includes more than one memory pattern. It is also possible that more than one configuration file are used to program a typical programmable logic device. The configuration files are usually stored in memory banks and are individually retrieved to program the corresponding programmable logic device. Typically, EPROMs are used to store the configuration files.

As the complexity of the programmable logic devices grows, so does the number of programmable elements used. This requires configuration files with a greater number of "1s" and "0s" to program the programmable memory devices. As the size of the configuration files increases, so does the size of the EPROMs needed to store them. Large EPROMs are expensive and require large silicon area to be manufactured. The size of the silicon area is more important when the EPROM is manufactured on the same substrate as the programmable logic device. The size of the EPROMs limit the complexity of the programmable logic device.

From the foregoing, it can be appreciated that there is a need for an apparatus and method of reducing the size of the configuration files before they are stored in memory, particularly where the configuration files are stored in memory located on the same substrate as the PLD.

SUMMARY OF THE INVENTION

The method and apparatus of the present invention compress the data file which is used to program a particular programmable logic device. The compressed data file is then stored in a memory element. Once it is necessary to program the programmable logic device, the compressed data file is decompressed and then is used to program the programmable logic device.

In one embodiment of the present invention, the compression and decompression are done by hardware. In a second embodiment, the compression is done by software external to the PL and the decompression is done by hardware either located on the PLD, or closely associated with the PLD and an external configuration file memory.

Therefore, the present invention offers a solution to the problems caused by storage of large memory patterns that are used in programming the programmable logic devices.

Other advantages of the present invention will be more evident as the invention is disclosed in the ensuing detailed description of the invention and claims.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** illustrates the preferred embodiment of the method of the present invention;

FIG. **2** is a flow chart showing the preferred method of performing the compression in the present invention;

FIG. **3** shows a circuit diagram of a compression circuit;

FIG. **4** shows the timing diagram of the circuit of FIG. **3** used to compress an example configuration file;

FIG. **5** is a flow chart which illustrates an alternative method in performing the compression in the present invention;

FIG. **6** shows one embodiment of the system according to the present invention;

FIG. **7** shows a second embodiment of the system according to the present invention;

FIG. **8** shows a third embodiment of the system according to the present invention;

FIG. **9** shows a fourth embodiment of the system according to the present invention; and

FIG. **10** shows a circuit diagram of a decompression circuit used in the present invention.

DESCRIPTION OF THE PREFERRED EMBODIMENTS

The present invention offers a solution to the problem of storing large configuration files. By reducing the number of bits in an individual configuration file through compression, the present invention eliminates the need for large memory elements either associated with or on the PLD in which to store them.

FIG. **1** illustrates a flow chart showing the steps of the method of the present invention. Process **100** first inputs the configuration file in step **102**. The configuration file is then compressed in step **104** using a software compression algorithm or a hardware compression circuit. The compression step reduces the number of bits in the configuration file by a ratio of as much as 2 to 1. The compressed configuration file is then stored in a memory element, such as a serial EPROM, in step **106**. Since the size of the generated configuration file is substantially smaller than the original configuration file, a memory element with smaller storage capacity can be used to store the generated file.

To program the programmable logic device, the original configuration file must be recreated. In step **108**, the compressed configuration file is decompressed using a decompression circuit. Finally, in step **110**, the configuration file generated in step **108** is used to program the programmable logic device.

In addition to reducing the size of the memory element, the reduction of the size of the individual configuration files allows for storing more than one configuration file in a single memory element. This eliminates the need for multiple memory elements where more than one configuration file can be used to program a programmable logic device.

As mentioned above, the compression step can be performed by software or hardware. The flow chart of FIG. 2 shows the preferred method of performing the compression step **104** in FIG. 1. The compression process shown in FIG. 2 begins by inputting a configuration file as shown in step **202**. This configuration file is typically stored in the memory of the host processing unit. This processing unit (shown in FIGS. **6** and **7**) executes the software realizing the steps shown in the flowchart FIG. 2. As mentioned above, a typical configuration file includes at least one memory pattern which is used to program the programmable logic device. However, almost always, configuration files include more than one memory pattern.

Process **200** continues by obtaining the first memory pattern to be compressed in step **204**. In step **205**, the value of a count variable is initialized to zero. The count variable is used to keep track of the number of low bits between two high bits in the memory pattern. As mentioned above, a low bit refers to a logic "0" bit and a high bit refers to a logic "1" bit. In step **206**, the first bit of information in the memory pattern is examined. If the first bit is a high bit, the process proceeds to step **214**, which will be described later.

If the first bit is a low bit, process **200** increments the value of the count variable and proceeds to step **210**. In step **210**, the next bit is examined. If the next bit is a high bit, process **200** proceeds to step **214**. If the next bit is a low bit, process **200** proceeds to step **212**. In step **212**, the value of the count variable is examined to determine whether it has reached a maximum preset value. If the value of the count variable is equal to the maximum value, then process **200** proceeds to step **214**, otherwise, it returns to step **208** and proceeds as explained above.

The maximum preset value of the count variable depends on the representation scheme used in step **214**. For example, in the embodiment of the present invention, a Binary Coded Hexadecimal Representation ("BCH") representation of the value of the count variable is generated at step **214**. It is well known in the art that four bits are used to generate the BCH representation of a single hexadecimal digit number. It is also well known that the maximum hexadecimal decimal number that can be represented by four bits is hexadecimal F or decimal fifteen. Therefore, in the present embodiment, the maximum preset value of the count variable is set to fifteen. Obviously, this number can change depending on the different schemes used to represent the value of the count variable.

Three different alternatives to reach step **214** are described above. While in step **214**, process **200** generates the BCH representation of the value of the count variable. The outcome of step **214** is stored in memory and will be retrieved later to construct the compressed configuration file. In step **216**, process **200** determines whether the last bit in the memory pattern was just examined. If the answer to step **216** is no, process **200** resets the value of the count variable to zero in step **212**, returns to step **210**, and proceeds as described above. If the answer to step **216** is yes, process **200** constructs the compressed memory pattern by retrieving and concatenating the BCH representations of the current memory pattern. The compressed memory pattern is stored in the memory of the host processor.

After step **218**, process **200** proceeds by determining whether the current memory pattern is the last memory pattern in the configuration file in step **220**. If the answer is no, process **200** retrieves the next memory pattern in the configuration file from the memory of the host processor, returns to step **205**, and continues as described above. If the answer to step **220** is yes, process **200** generates the compressed version of the configuration file in step **224**. This is done by retrieving the compressed memory patterns from the memory of the host processor and forming a file from all compressed memory patterns.

FIG. **3** illustrates an electronic circuit **300** which performs the compression step **104** in FIG. **1** using hardware. Circuit **300** includes a counter **302**, 4-bit registers **304** and **306**, "D" flip-flop ("FF") **308**, NOR gates **310** and **312**, 4-input AND gate **314**, AND gate **316**, OR gate **318**, and invertors **320** and **334**.

Counter **302** is used to count the number of low bits between two high bits in a memory pattern. Counter **302** includes four inputs A–D, a LOAD input which is asserted when connected to a low level signal, a clock input "CK", and four outputs **328**–**332**. It should be realized that a low level signal is realized by providing ground potential at the prospective inputs. The information on inputs A–D when loaded in counter **302** provides the initial count value of counter **302**. In operation, counter **302** counts up or down from this initial count value. For example, if inputs A and B carry a low level signal and inputs C and D carry a high level signal, the initial value of counter **302** is three. In the embodiment of FIG. **3**, inputs A–D are permanently connected to a low level signal. Outputs **328**–**331** carry the binary equivalent of the latest count value of the counter. Inputs A–D are loaded in counter **302** when a low level signal appears on LOAD input **332**.

Outputs **328**–**331** are connected to inputs 1D–4D of both 4-bit registers **304** and **306**. This allows for latching the information on outputs **328**–**331** in registers **304** and **306**. Each of the two registers **304** and **306** further includes a clear input ("CLR"), a clock input and other inputs that are connected to ground potential as shown in FIG. **3**. The CLR input of both registers **304** and **306** are connected to GRESET signal **322**. GRESET input is used to initialize circuit **300**, as will be described later. The CK input of register **304** is connected to the inverse Q output of FF **308** and the CK input of register **306** is connected to the Q output of FF **308**.

The D input of FF **308** is connected to its inverse Q output and its clock input is connected to the output of AND gate **316**. The signal at the output of AND gate **316** is also referred to as the GRAB signal **336**. As it will be more clearly described below, a high to low or a low to high transition of the GRAB signal is used to latch the information on outputs **328**–**331**. FF **308** further includes a CLR input which is connected to the output of invertor **334**. Invertor **334** generates a clear signal to clear FF **308** by inverting the GRESET signal **322**. The A input of AND gate **316** is connected to the output of OR gate **318**, and the B input of AND gate **316** is connected to the output of invertor **320**. The input of invertor **320** is connected to DATACLK signal **326**, which is the system clock signal. The A input of OR gate **318** is connected to the output of AND gate **314** and its B input is connected to the DATA input terminal **324**. The individual bits of each memory pattern serially enters compression circuit **300** via DATA input terminal **324**.

The inputs of AND gate **314** are connected to outputs **328**–**331**. DATA input terminal **324** and DATACLK signal **326** are also connected to A and B inputs of NOR gate **312**, respectively. The A and B inputs of NOR gate **310** are connected to the GRESET signal **322** and GRAB signal **336**, respectively.

Circuit **300** operates as follows. A high level signal connected to GRESET signal **322** forces NOR gate **310** to generate a low signal at its output. This places a low level signal at the LOAD input of counter **302**, which loads in the information on its inputs. GRESET signal **322** is also used to clear registers **304** and **306**, and FF **308** via invertor **334**. With FF **308** cleared, a low level signal appears at its Q output and a high level signal appears at its inverse Q output.

To begin the compression, a low level signal is applied to GRESET **322**. As mentioned before, the information bits in the memory pattern serially enters circuit **300** via DATA input terminal **324**. As long as low level bits are inputted, the output of NOR gate **312** follows the clock input at DATACLK terminal **326**. This provides a clock signal at the CK input of counter **302** which forces counter **302** to count. In the embodiment of FIG. **3**, counter **302** counts up from the initial count value. Thus, so far, the counter is counting the number of low level bits in the memory pattern.

Once the first high level bit appears at DATA input terminal **324**, the output of NOR gate **312** is forced to a low level. This stops the clock signal at the CK input of counter **302**, thus stopping counter **302** from further counting. A high level bit at DATA input terminal **324** also forces the output of OR gate **318** to high, which forces the output of AND gate **316** to follow the clock signal at the output of invertor **320**. Invertor **320** inverts the clock signal received at DATACLK terminal **326**.

As mentioned above, the output of AND gate **316** is connected to the CLK input of FF **308**, and the inverse Q of FF **308** is set to high once FF **308** is cleared. This places a high level signal at the D input of FF **308**. Therefore, the first high to low transition of the clock signal at the output of AND gate **316** forces the Q output of FF **308** to high level. A transition from low to high at the Q output of FF **308** provides a clock signal for register **306**, thus causing the information on outputs **328**–**331** to be latched in register **306**.

The above occurs during a single high to low transition of the signal at the output of AND gate **316**. This signal is referred to as the GRAB signal. The next transition from a low to a high level at the output of AND gate **316** forces the output of NOR gate **310** to low, which provides a load signal to counter **302**. Counter **302** is then reloaded with the initial value of zero.

The next time that a high bit is detected, again the output of OR gate **318** switches to high level, which forces the output of AND gate **316** to follow the clock signal at the output of invertor **320**. This time, a transition from low to high at the output of AND gate **316** forces the Q output of FF **308** to a low level, and low to high at the inverse Q of FF **308** provides a clock signal for register **304**, which allows the information on outputs **328**–**331** to be latched into register **304**. This is the second GRAB signal.

Above, we described how the GRAB signals are generated if a high bit is detected after a low bit. If the number of low bits between two high bits are more than a present maximum number, circuit **300** generates a GRAB signal. If the counter counts to the maximum value, then outputs **328**–**331** will all carry high level signals. This forces the output of AND gate **314** to high, which forces the output of OR gate **318** to high. The rest of the operation is similar to what has been described above. In the embodiment of FIG. **3**, the maximum count is fifteen. Therefore, if there are more than fifteen logic "0s" between two logic "1s" in the memory pattern, a GRAB signal is generated by circuit **300**.

FIG. **4** shows a timing diagram of circuit **300** compressing the following memory pattern:

00010011000000000000000011

Signal A represents the above memory pattern, signal B represents the clock signal at DATACLK terminal **326**, signal C represents the signal at the output of AND gate **316** (i.e. the GRAB signal), signal D represents the clock signal to counter **302**, signals E and F represent clock signals to registers **304** and **306**, signal G represents the signal at GRESET terminal **322**, signal H represents the output of counter **302**, and signal I represents the output of circuit **300**.

As explained above, two GRAB signals are needed to load both registers **304** and **306**. Therefore, valid compressed data is read on the falling edge of every other GRAB signal. Arrows **340** point to the position in signal C where valid signals are read. In the present example, counter **302** counts the number of low bits, which is three, until the first high bit is detected. Then, the binary value of three is loaded in register **306** as explained above with reference to FIG. **3**. Since the first high bit has been detected, a zero initial count is reloaded in counter **302**. The next low bit in the memory pattern forces counter **302** to start counting again. There are two other low bits in the data before the next high bit is detected.

Once the next high bit is detected, outputs QA–QD are loaded in register **304** as explained above. At this point, the outputs of register **304** and **306** are read and stored (not shown). The detected high bit forces a reloading of counter **302**. The next bit is another high bit which forces the generation of another GRAB signal **342**. This forces the loading of the information on outputs **328**–**331**, which is the binary equivalent of zero, in register **306**. This process continues until the whole memory pattern is compressed. The following shows the results of the example in FIG. **4**.

Pattern: 00010011000000000000000011

Compressed: 001100100000111100100000Note that the first four bits of the pattern "0001" are compressed as the BCH digit "3" or "0011" signifying three "0's" followed by a "1". The next three bits of the pattern "001" are compressed into the hexadecimal digit "2" or "0010" signifying two "0's" preceding a "1" bit. The next bit in the pattern is compressed as hexadecimal "0000" signifying no "0's" preceding a "1" bit, and bits 9 through 23 of the pattern are compressed as hexadecimal F or "1111" signifying a string of fifteen "0's" with no "1" bits It can be seen from the above that the compressed pattern has fewer bits than the un-compressed pattern. The above is for illustrative purposes only.

FIG. **5** shows an alternative method of performing the compression step **104** in FIG. **1**. The compression process, according to the embodiment of FIG. **4**, starts by obtaining the configuration file to be compressed in step **402**. Next, process **400** selects the first memory pattern to be compressed, initializes a test register by loading a logic "0" in each individual register of the test register in step **404**. The test register is used to temporarily store the compressed memory pattern and includes a fairly large number of individual registers. Each individual register holds one bit of compressed data.

In steps **406** and **408**, the first four bits of the memory pattern are examined. If all are low level bits, the first bit of the test register is set to low in step **410** and the process proceeds to execute step **414**. If any of the four bits is a high level bit, a leading high bit is added to the above four bits and the resulting five bits are stored in the first five registers of the test register in step **412**. The process then proceeds to step **414**.

In step **414**, the compressor determines whether the four bits of data just examined were the last four bits in the memory pattern. If the answer is no, the next four bits of the memory pattern are examined in step **418**. Again, if all are low level bits, process **400** sets the next bit of the test register to low level in step **422** and returns to step **414**. If any of the four bits is a high level bit, process **400** adds a leading high level bit to the examined four bits and stores the resulting five bits in the next five bits of the test register. Process **400** then returns to step **414**. Once in step **414**, the compressor proceeds as explained above.

If the answer to the question in step **414** is yes, which means that the end of the memory pattern is reached, process **400** proceeds to step **424**. In step **424**, process **400** stores the content of the test register, which is the compressed version of the memory pattern. In step **426**, process **400** determines whether the previous memory pattern was the last memory pattern in the configuration file. If no, process **400** selects the next memory pattern in step **428**, re-initializes the test register in step **432**, and proceeds to step **406**. If the answer to the step **426** is yes, the compression is over. The resulting configuration file is then stored in the serial EPROM, as shown in step **106** of FIG. **1**.

As mentioned before, the compression can be performed by software or hardware. In the former, the software tool residing in a host computer performs the compression step. In the latter, the compression step is performed by an electronic circuit, such as the one shown in FIG. **3**. The electronic circuit can reside in the host computer or it can be packaged with the memory element used to store the compressed configuration files.

FIG. **6** shows an embodiment of the system of the present invention where software is used to perform the compression step **104** in FIG. **1**. FIG. **6** shows a host computer **502**, compression software **504**, serial device integrated circuit **506** which includes memory array **507** and decompression circuit **508**, and programmable logic device **510**. Host computer **502** can be any personal computer, such as IBM or IBM compatible personal computers, Macintosh personal computers, SUN or DEC (Digital Equipment Corporations) work stations.

In operation, the user generates one or more configuration files using a programming tool (not shown), such as the ALTERA MAX™ programming tools, supplied by the manufacturer of programmable logic device **510**. Hereinafter, we will assume that only one configuration file is generated by the user. Compression software **504** can be a part of the programmable tool, or it can be a separate software tool used by the programming tool. Compression software **504** compresses the configuration file and generates a compressed configuration file. The compressed configuration file is then stored in memory array **507** of serial device **506**. To program programmable logic device **510**, the programming tool retrieves the stored configuration file from memory array **507** by addressing the memory locations where the compressed configuration file is stored. Since the output of memory array **507** is connected to the input of decompression circuit **508**, the retrieved file is inputted into decompression circuit **508**. Decompression circuit **508** decompresses the compressed configuration file and regenerates the original configuration file. The original configuration file is then used to program programmable logic device **510**.

FIG. **7** shows the embodiment of the present invention wherein the compression is done by hardware. The system of FIG. **7** includes a host computer **502**, a serial device integrated circuit **506**, and a programmable logic device **510**. Serial device integrated circuit **506** includes compression circuit **300** (refer to FIG. 3), memory array **507**, and decompression circuit **508**. Host computer **502** is of the type mentioned above.

In operation, the user generates a configuration file using the above-mentioned programming tool. The generated configuration file is then sent to serial device integrated circuit **506** to be stored. However, before storing the configuration file, it is compressed by compression circuit **300**. The compressed configuration file is then stored in memory array **507**. To program programmable logic device **510**, the programming tool retrieves the stored configuration file from memory array **507** by addressing the memory locations where the compressed configuration file is stored. Since the output of memory array **507** is connected to the input of decompression circuit **508**, the retrieved file is inputted into decompression circuit **508**. Decompression circuit **508** decompresses the compressed configuration file and regenerates the original configuration file. The original configuration file is then used to program programmable logic device **510**.

FIGS. **6** and **7** illustrate the situations where the decompression circuit is part of serial device integrated circuit **506**. FIGS. **8** and **9** illustrate the embodiments of the present invention where decompression circuit **508** is part of programmable logic device **510**.

FIG. **8** shows a host computer **502**, compression software **504**, serial device integrated circuit **506** which includes memory array **507**, and programmable logic device **510** which includes decompression circuit **508**.

FIG. **9** shows a host computer **502**, a serial device integrated circuit **506**, and a programmable logic device **510** which includes decompression circuit **508**. Serial device integrated circuit **506** includes compression circuit **300** (refer to FIG. 3) and memory array **507**.

As mentioned above, host computer **502** can be any personal computer, such as IBM or IBM compatible personal computers, Macintosh personal computers, SUN or DEC (Digital Equipment Corporations) work stations. The systems of FIGS. **8** and **9** operate exactly as do the systems in FIGS. **6** and **7**, respectively, except that decompression circuit **508** now resides within the package of programmable logic device **510**.

FIG. **10** illustrates an embodiment of decompression circuit **508** of FIGS. **6** and **7**. FIG. **10** shows a 4-bit down counter **520**, a 3input NOR gate **524**, a 4-input NAND gate **523**, and a "D" FF **526**. Decompression circuit **508** operates on four parallel bits of the compressed configuration file at a time and generates output bits in response to the information on its four inputs.

As shown in FIG. **10**, counter **520** includes four inputs which are connected to input lines X0–X3, a clock input **521** coupled to the system clock **530**, and a SyncLoad input **522** which is connected to the output of NOR gate **524**. Counter **520** further includes four outputs connected to the inputs of NOR gate **524**. Input lines X0–X3 are also connected to the inputs of NAND gate **523**. FF **526** includes a "D" input which is connected to the output of NOR gate **524**, a clock input **527** which is connected to the system clock **530**, a clear ("CLR") input **528** which is connected to the output of NAND gate **523**, and a Q output.

The operation of circuit **508** is best understood through an example. Assume that input lines X0–X3 carry the BCH equivalent of the number five. As describe above, a four-bit binary equivalent of number five represents the situation where five low level bits were detected before a high level bit was detected in a memory pattern. Therefore, circuit **508** must generate six output bits. The first five bits are low level bits and the last bit is a high level bit. Also, assume that circuit **508** is initialized which means that the outputs of counter **520** are all set at low level. This generates a high level signal at the output of NOR gate **524**, which is connected to SyncLoad input **522**. As long as a high level signal is connected to SyncLoad input **522**, the information on the input of counter **520** is loaded in as the initial count value. In the present example, the binary equivalent of the number five will be loaded in counter **520**.

This immediately forces the output of NOR gate **524** to low. Since a continuous clock signal is connected to the clock input of FF **526**, the next clock signal forces the output of FF **526** to low. Thus, a first low level bit is then generated. On the next clock signal the counter decrements its count value. A binary equivalent of the new count value appears on the outputs of counter **320**. The output of NOR gate **524** remains at low level, which provides a low signal at the "D" input of FF **526**. The next clock input to FF **526** will not change the level of its output, maintaining a low level at the output of FF **526**. Thus, the second low level bit is generated.

This process continues for the next three clock signals. At the end of the third clock signal, the outputs of counter **520** carry the binary equivalent of zero. This forces the output of NOR gate **524** to high level, which places a high level at the output of FF **526**. The next clock signal switches the output of FF **526** from low to high. Thus, the sixth bit, which is a high level bit, is generated. The above process continues until the end of the compressed configuration file is reached.

NAND gate **523** is provided for situations where X0–X3 are all high, which means that at least fifteen low bits were detected between two high bits. An all high level input to NAND gate **523**, forces its output to low level. This provides a valid clear signal of FF **526** to the CLR input, which forces the Q output of FF **526** to remain low as long as X0–X3 are all high. Thus, for fifteen clock signals the output of FF **526** remains at low level. After fifteen clock signals, the next four bits in the compressed configuration file is provided to X0–X3 inputs. If any of the four inputs X0–X3 carry a low bit, the output of NAND gate **523** switches to high level, and circuit **508** functions as explained above.

The present invention has now been explained with reference to specific embodiments. Other embodiments will be apparent to those of ordinary skill in the art. It is therefore not intended that this invention be limited except as indicated by the appended claims.

What is claimed is:

1. A method of programming a programmable logic device, comprising:

compressing a configuration file by instructions in a digital computer to generate a compressed configuration file;

storing said compressed configuration file into a storage device used for programming said programmable logic device;

decompressing said compressed configuration file to generate a set of configuration data; and

programming the programmable logic device using said set of configuration data, wherein said storage device comprises electronically programmable read only memory for storing said compressed configuration file and

wherein said programmable logic device comprises an electronic decompression circuit to perform said decompression step.

2. The method of claim **1**, wherein said compressing step comprises:

initializing a value of a count variable;

counting a number of first logic level bits in said input file before detecting a second logic level bit;

setting said value of said count variable equal to the result of said counting step; and

generating a representation of said value of said count variable.

3. The method of claim **2** further comprising the steps of:

counting said first logic level bits before detecting a second one of said second logic level bit;

performing said setting and said generating steps; and

re-initializing said value of said count variable.

4. The method of claim **3** further comprising generating said compressed configuration file from said representations of the value of said count variable.

5. The method of claim **4**, wherein said compressed configuration file is generated by concatenating said representations.

6. The method of claim **2**, wherein generating a representation of said count variable comprises generating a binary-coded hexadecimal equivalent of said value.

7. The method of claim **1**, wherein said compressing step is done by an electronic compression circuit.

8. The method of claim **7**, wherein said storage device comprises:

said electronic compression circuit;

electronically programmable read only memory for storing said compressed configuration file; and

an electronic decompression circuit to perform said decompression step.

9. The method of claim **1**, wherein said configuration file is larger than said compressed configuration file.

10. The method of claim **1**, wherein said set of configuration data is equal in size to said configuration file.

11. The method of claim **1**, wherein said decompression step is performed using a run-length decompression scheme.

12. A method of programming a programmable logic device, comprising:

compressing a configuration file to generate a compressed configuration file by instructions in a digital computer;

storing said compressed configuration file into a dedicated storage device used for programming said programmable logic device

decompressing said compressed configuration file to generate a set of configuration data; and

programming the programmable logic device using said set of configuration data

wherein said storage device comprises:

an array of electronically programmable read only memory for storing said compressed configuration file; and

an electronic decompression circuit to perform said decompression step.

13. The method of claim **12**, wherein said compressing step comprises

initializing a value of a count variable;

counting a number of first logic level bits in said input file before detecting a second logic level bit

setting said value of said count variable equal to the result of said counting step; and

generating a representation of said value of said count variable.

14. The method of claim **13** further comprising the steps of:

counting said first logic level bits before detecting a second one of said second logic level bit;

performing said setting and said generating steps; and re-initializing said value of said count variable.

15. The method of claim **14** further comprising generating said compressed configuration file from said representations of the value of said count variable.

16. The method of claim **15**, wherein said compressed configuration file is generated by concatenating said representations.

17. The method of claim **13**, wherein generating a representation of said count variable comprises generating a binary-coded hexadecimal equivalent of said value.

18. The method of claim **12**, wherein said compressing step is done by software in a digital computer.

19. The method of claim **12**, wherein said compressing step is done by an electronic compression circuit.

20. The method of claim **12**, wherein said configuration file is larger than said compressed configuration file.

21. The method of claim **12**, wherein said set of configuration data is equal in size to said configuration file.

22. A method of programming a programmable logic device, comprising:

compressing a configuration file to generate a compressed configuration file using an electronic compression circuit;

storing said compressed configuration file into a dedicated storage device used for programming said programmable logic device;

decompressing said compressed configuration file to generate a set of configuration data; and

programming the programmable logic device using said set of configuration data wherein said storage device comprises said electronic compression circuit and electronically programmable read only memory for storing said compressed configuration file; and

said programmable logic device comprises an electronic decompression circuit to perform said decompression step.

23. A method of programming a programmable logic device comprising the steps of:

compressing a data file before storing the file in a configuration memory;

compressing said data and storing compressed data in said first integrated circuit;

decompressing said compressed data to form a decompressed data stream; and

programming the programmable logic device with said decompressed data stream.

24. An apparatus to be used in programming a programmable logic device, comprising:

compression means to compress an input file and to generate a first output file;

a storage device coupled to said compression means for receiving and storing said first output file;

decompression means on said storage device for decompressing said first output file to generate a second output file; and

means to transfer said second output file to the programmable logic device.

25. Apparatus as in claim **24**, wherein said input file is larger than said first output file.

26. Apparatus as in claim **24**, wherein said input file and said second output file are equal in size.

27. Apparatus as in claim **24**, wherein said compression means comprises a software compression program running on a development computer.

28. Apparatus as in claim **24**, wherein said compression means comprises a compression circuit associated with a development computer.

29. Apparatus as in claim **24**, wherein said decompression means comprises a run-length decompression circuit.

30. Apparatus for run length compressing a serial data file having a plurality of first and second logic level bits wherein said apparatus requires only a single data clock, comprising:

input terminal for inputting said serial data file;

clock terminal for inputting said data clock;

a counter having a plurality of inputs and outputs, said inputs connected so that said counter begins counting from zero when a reset signal is applied;

a plurality of parallel-in registers with inputs coupled to outputs of said counter, and outputs generating a compressed data file;

a logic gate with its inputs coupled to the outputs of said counter for generating a maximum signal indicating that said counter has reached its maximum count; and

a first logic circuit for generating a grab signal with a first input coupled to said input terminal, a second input coupled to said clock terminal, a third input coupled to said maximum signal and a first output coupled to a clock input of said registers.

31. The apparatus of claim **30**, wherein said counter counts a number of said first logic level bits in said data file before one of said second logic level bits is detected by said logic circuit.

32. The apparatus of claim **30**, wherein said plurality of registers comprise first and second registers and further comprising:

a flip-flop with a clock input coupled to said grab signal, an first output coupled to a clock input of said first register and a second output coupled to a clock input of said second register.

33. The apparatus of claim **32**, wherein said first register latches the output of said counter in response to said first clock signal.

34. The apparatus of claim **32**, wherein said second register latches the output of said counter in response to said second output.

* * * * *

Exhibit C

US007036046B2

(12) **United States Patent**
Rally et al.

(10) **Patent No.:** **US 7,036,046 B2**
(45) **Date of Patent:** **Apr. 25, 2006**

(54) **PLD DEBUGGING HUB**

(75) Inventors: **Nicholas James Rally**, San Mateo, CA (US); **Alan Louis Herrmann**, Sunnyvale, CA (US)

(73) Assignee: **Altera Corporation**, San Jose, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 654 days.

(21) Appl. No.: **10/295,265**

(22) Filed: **Nov. 14, 2002**

(65) **Prior Publication Data**

US 2004/0098638 A1 May 20, 2004

(51) **Int. Cl.**
G06F 11/00 (2006.01)

(52) **U.S. Cl.** **714/39**; 714/725

(58) **Field of Classification Search** 714/39, 714/30, 725, 733, 734; 703/15, 16, 17, 28; 326/38, 39
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,527,234 A		7/1985	Bellay
4,696,004 A		9/1987	Nakajima
4,788,492 A		11/1988	Schubert
4,835,736 A		5/1989	Easterday
4,847,612 A		7/1989	Kaplinsky
4,873,459 A		10/1989	El Gamo et al.
5,036,473 A		7/1991	Butts et al.
5,058,114 A		10/1991	Kuboki et al.
5,124,588 A		6/1992	Baltus et al.
5,157,781 A		10/1992	Harwood et al.
5,329,470 A		7/1994	Sample et al.
5,365,165 A		11/1994	El-Ayat et al.
5,425,036 A	*	6/1995	Liu et al. 714/35
5,452,231 A		9/1995	Butts et al.
5,568,437 A		10/1996	Jamal
5,572,712 A		11/1996	Jamal
5,629,617 A		5/1997	Uhling et al.
5,640,542 A		6/1997	Whitsel et al.
5,661,662 A		8/1997	Butts et al.

(Continued)

FOREIGN PATENT DOCUMENTS

EP 0 762 279 A1 3/1997

OTHER PUBLICATIONS

Marantz, Joshua, "Enhanced Visibility and Performance on Functional Verification by Reconstruction", Proceedings of the 35th Annual Conference on Design Automation Conference, pp. 164-169. 1998.

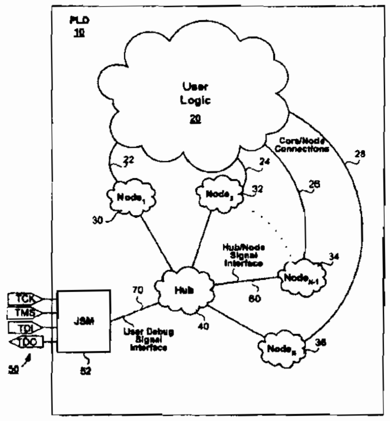
(Continued)

Primary Examiner—Scott Baderman
(74) *Attorney, Agent, or Firm*—Beyer Weaver & Thomas, LLP

(57) **ABSTRACT**

User logic within a PLD is debugged by way of the hub. The PLD includes a serial interface (such as a JTAG port) that communicates with a host computer. Any number of client modules are within the PLD and provide instrumentation for the PLD. A module is a logic analyzer, fault injector, system debugger, etc. Each client module has connections with the user logic that allows the instrumentation to work with the user logic. The hub communicates with each client module over a hub/node signal interface and communicates with the serial interface over a user signal interface. The hub routes instructions and data from the host computer to a client module (and vice-versa) via the serial interface and uses a selection identifier to uniquely identify a module. The hub functions as a multiplexor, allowing any number of client modules to communicate externally though the serial interface as if each node were the only node interacting with user logic.

23 Claims, 9 Drawing Sheets



U.S. PATENT DOCUMENTS

5,717,695	A	2/1998	Manela et al.	
5,717,699	A	2/1998	Haag et al.	
5,764,079	A	6/1998	Patel et al.	
5,821,771	A	10/1998	Patel et al.	
5,870,410	A *	2/1999	Norman et al.	714/25
5,960,191	A	9/1999	Sample et al.	
5,983,277	A	11/1999	Heile et al.	
6,014,334	A	1/2000	Patel et al.	
6,016,563	A	1/2000	Fleisher	
6,020,758	A	2/2000	Patel et al.	
6,104,211	A *	8/2000	Alfke	326/91
6,107,821	A	8/2000	Kelem et al.	
6,157,210	A	12/2000	Zaveri et al.	
6,182,247	B1 *	1/2001	Herrmann et al.	714/39
6,212,650	B1	4/2001	Guccione	
6,223,148	B1	4/2001	Stewart et al.	
6,247,147	B1 *	6/2001	Beenstra et al.	714/39
6,259,271	B1	7/2001	Couts-Martin et al.	
6,286,114	B1 *	9/2001	Veenstra et al.	714/39
6,317,860	B1 *	11/2001	Heile	716/5
6,321,369	B1 *	11/2001	Heile et al.	716/11
6,389,558	B1 *	5/2002	Herrmann et al.	714/39
6,460,148	B1 *	10/2002	Veenstra et al.	714/39
6,481,000	B1 *	11/2002	Zaveri et al.	716/17
6,704,889	B1 *	3/2004	Veenstra et al.	714/39
6,754,862	B1 *	6/2004	Hoyer et al.	714/725
6,794,896	B1 *	9/2004	Brebner	326/38
6,891,397	B1 *	5/2005	Brebner	326/41
2003/0110430	A1 *	6/2003	Bailis et al.	714/725
2004/0032282	A1 *	2/2004	Lee et al.	326/39

OTHER PUBLICATIONS

Stroud, Charles et al., "Evaluation of FPGA Resources for Built-in-Self-test of Programmable Logic Blocks", Proceedings of the 1996 ACM 4th International Symposium on Field-programmable Gate Arrays, p. 107. 1996.

Collins, Robert R., "Overview of Pentium Probe Mode", (www.x86.org/ariticles/problemd/ProbeMode.htm), Aug. 21, 1998, 3 pgs.

Collins, Robert R., "ICE Mode and the Pentium Processor", (www.x86.org/ddj/Nov97/Nov97.htm), Aug. 21, 1986, 6 Pgs.

"PentiumPro Family Developer's Manual", vol. 1: Specifications, Intel®Corporation, 1996, 9 Pgs.

"Pentium® Processor User's Manual", vol. 1, Intel®Corporation, 1993, Pgs. 3-11.

Xilinx, Inc.; ISE Logic Design Tools: ChipScope.

Praveen K. Jaini and Nur A. Touba; "Observing Test Response of Embedded Cores through Surround Logic"; 1999 IEEE.

Nur A. Touba and Bahram Pouya; "Testing Embedded Cores Using Partial Isolation Rings"; 1997 IEEE.

* cited by examiner

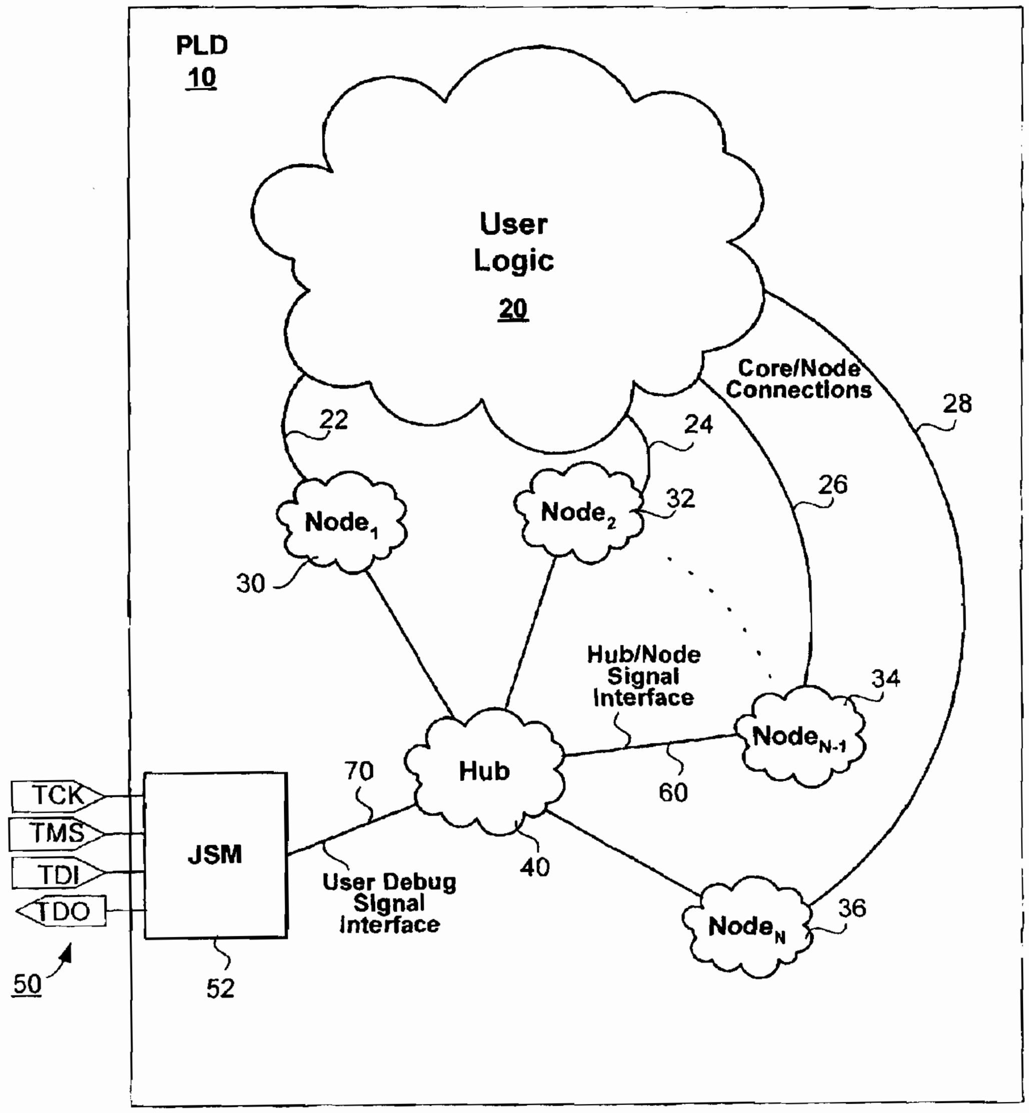

PLD
10
User Logic
20
Core/Node Connections
22
24
26
28
Node$_1$
Node$_2$
30
32
Hub/Node Signal Interface
Node$_{N-1}$
34
Hub
60
70
TCK
TMS
TDI
TDO
JSM
User Debug Signal Interface
40
Node$_N$
36
50
52

FIG. 1

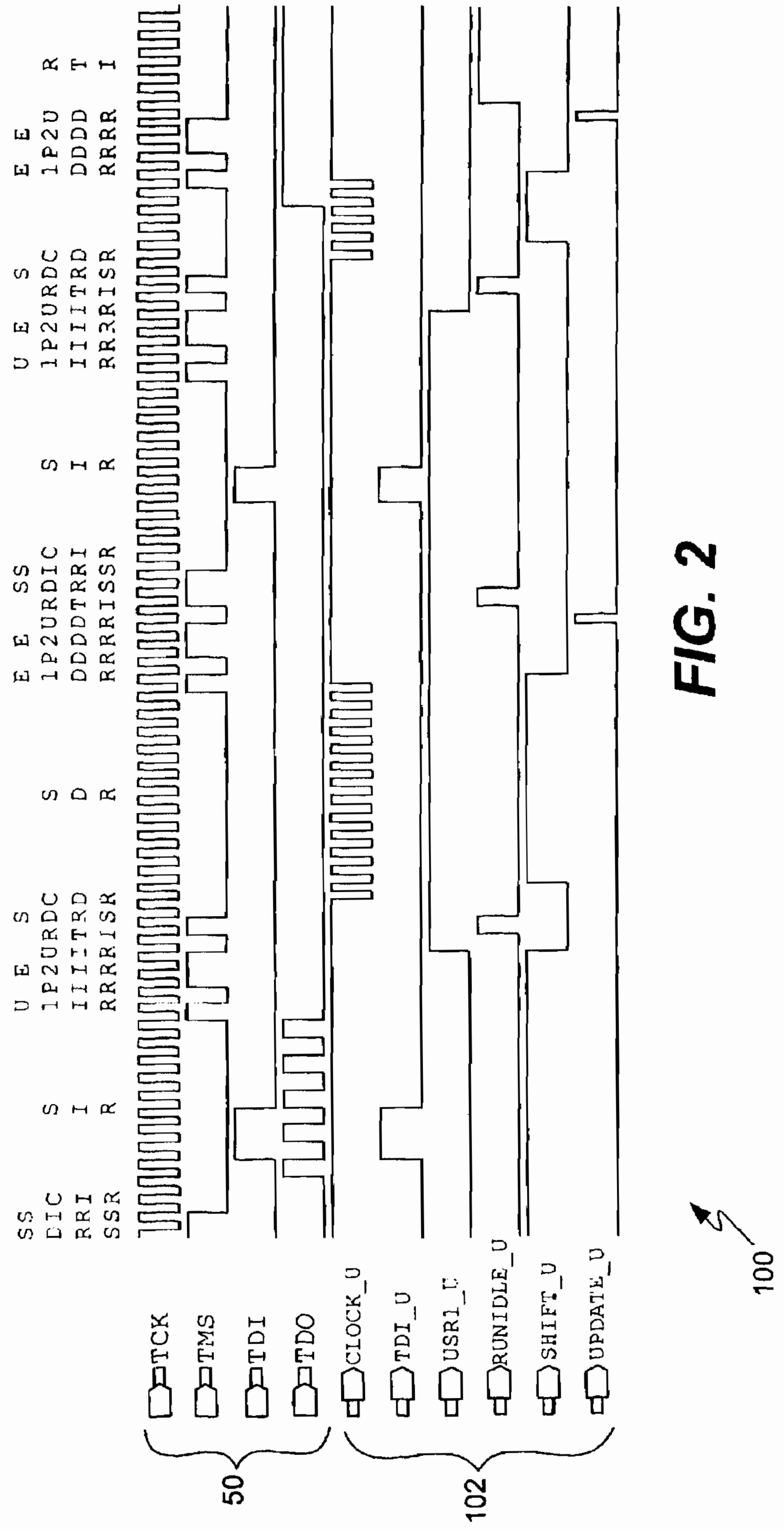

TCK
TMS
TDI
TDO
CLOCK_U
TDI_U
USR1_U
RUNIDLE_U
SHIFT_U
UPDATE_U
50
102
100

FIG. 2

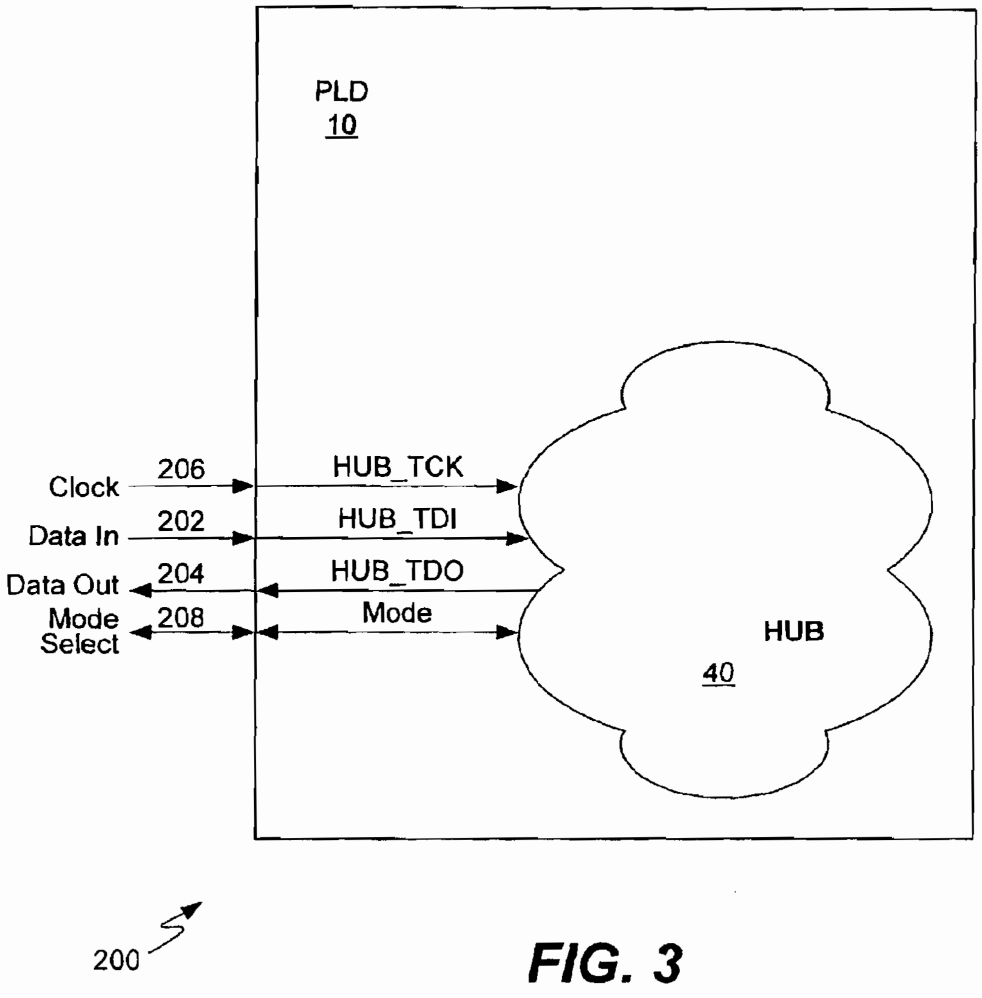
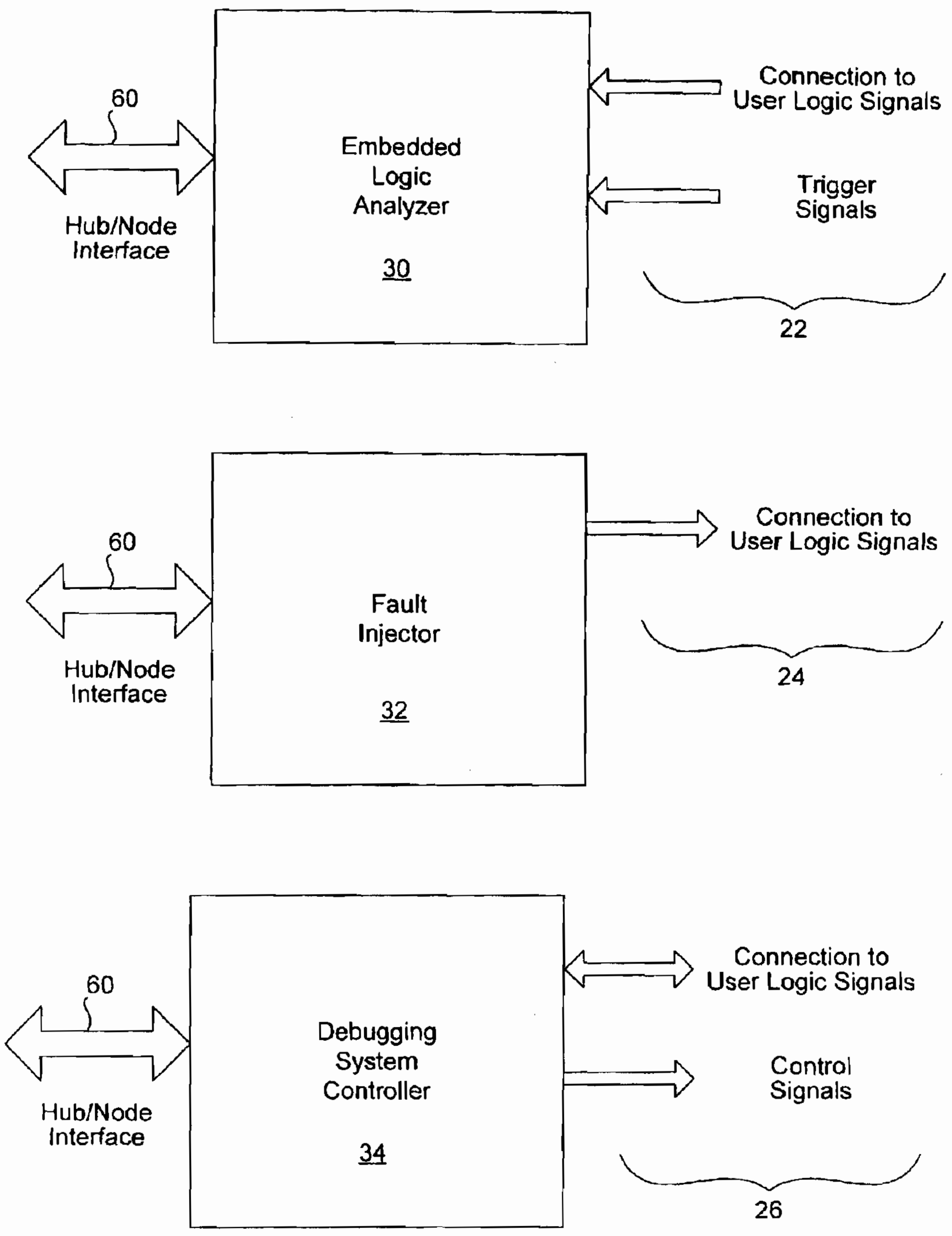

PLD
10
Clock
206
HUB_TCK
Data In
202
HUB_TDI
Data Out
204
HUB_TDO
Mode Select
208
Mode
HUB
40
200

FIG. 3



Node Examples

FIG. 4

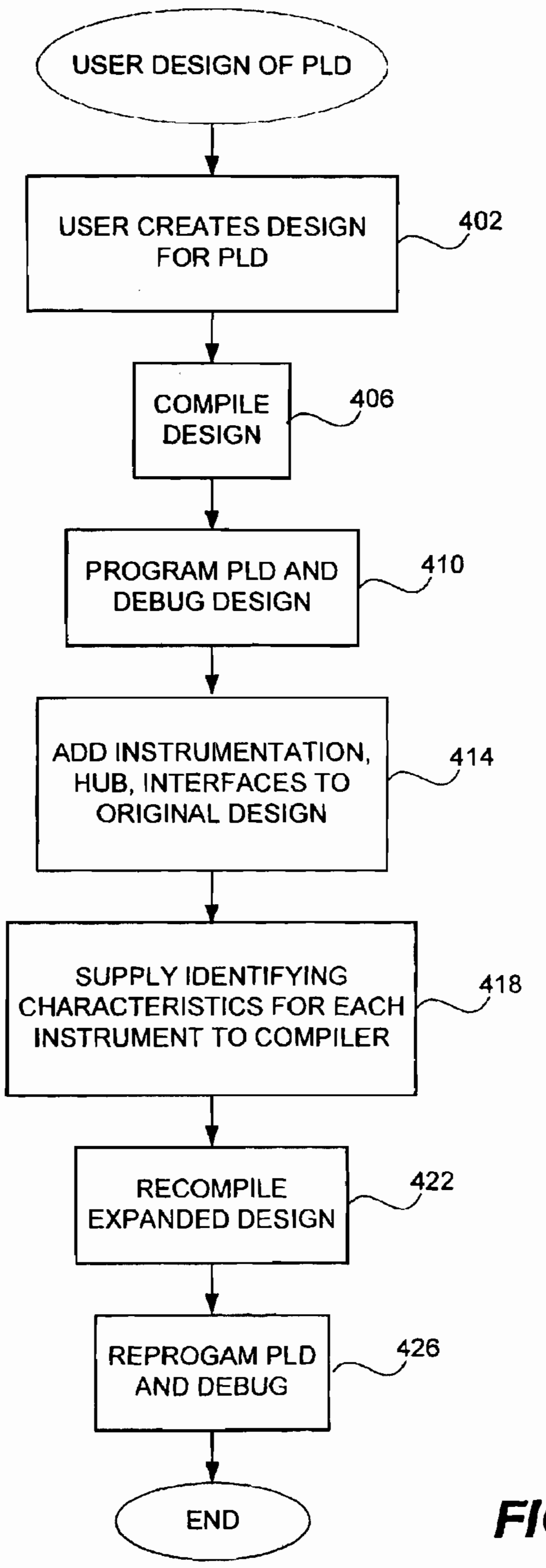

USER DESIGN OF PLD
USER CREATES DESIGN FOR PLD
402
COMPILE DESIGN
406
PROGRAM PLD AND DEBUG DESIGN
410
ADD INSTRUMENTATION, HUB, INTERFACES TO ORIGINAL DESIGN
414
SUPPLY IDENTIFYING CHARACTERISTICS FOR EACH INSTRUMENT TO COMPILER
418
RECOMPILE EXPANDED DESIGN
422
REPROGAM PLD AND DEBUG
426
END

FIG. 5

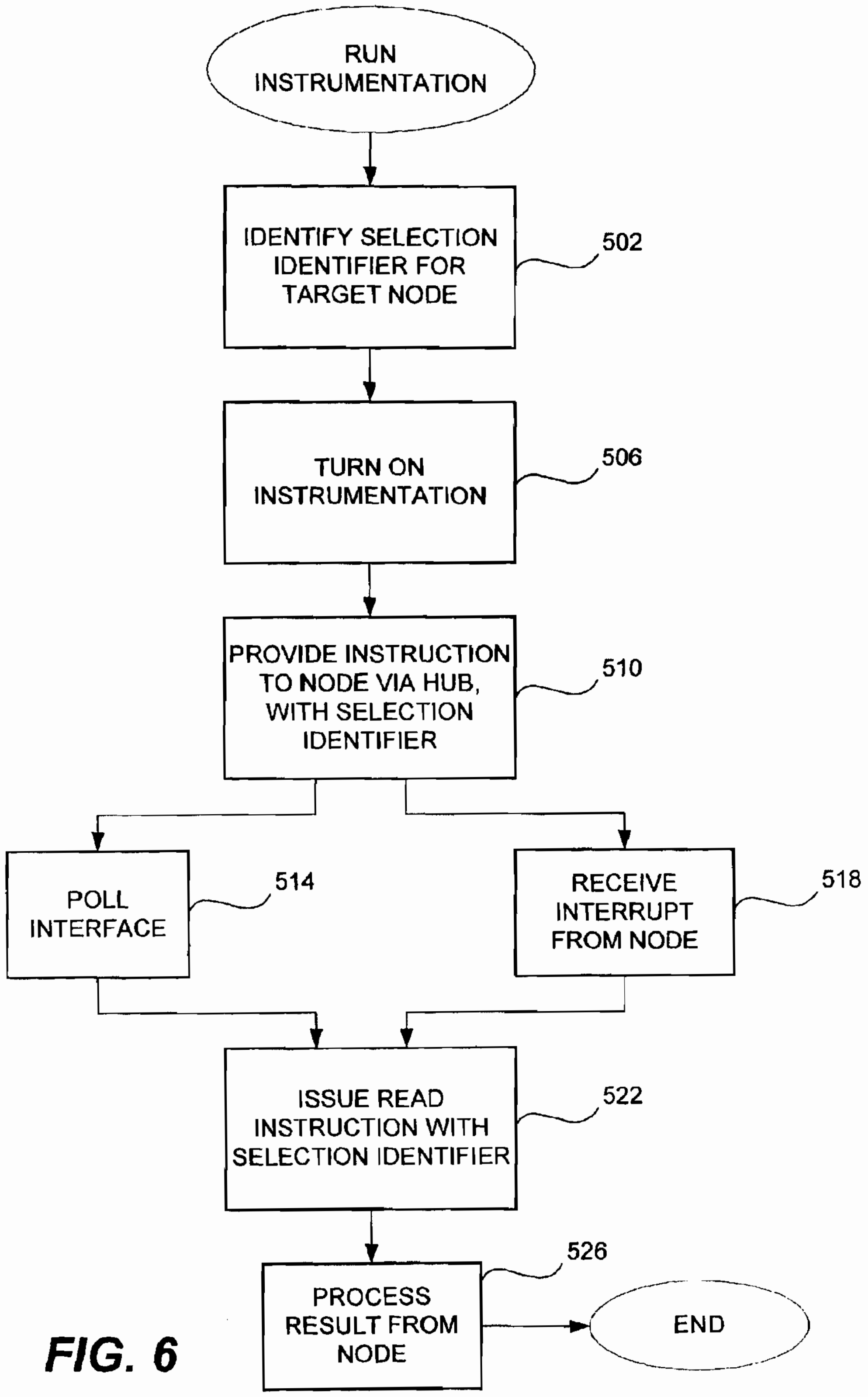

RUN INSTRUMENTATION
IDENTIFY SELECTION IDENTIFIER FOR TARGET NODE
502
TURN ON INSTRUMENTATION
506
PROVIDE INSTRUCTION TO NODE VIA HUB, WITH SELECTION IDENTIFIER
510
POLL INTERFACE
514
RECEIVE INTERRUPT FROM NODE
518
ISSUE READ INSTRUCTION WITH SELECTION IDENTIFIER
522
PROCESS RESULT FROM NODE
526
END

FIG. 6

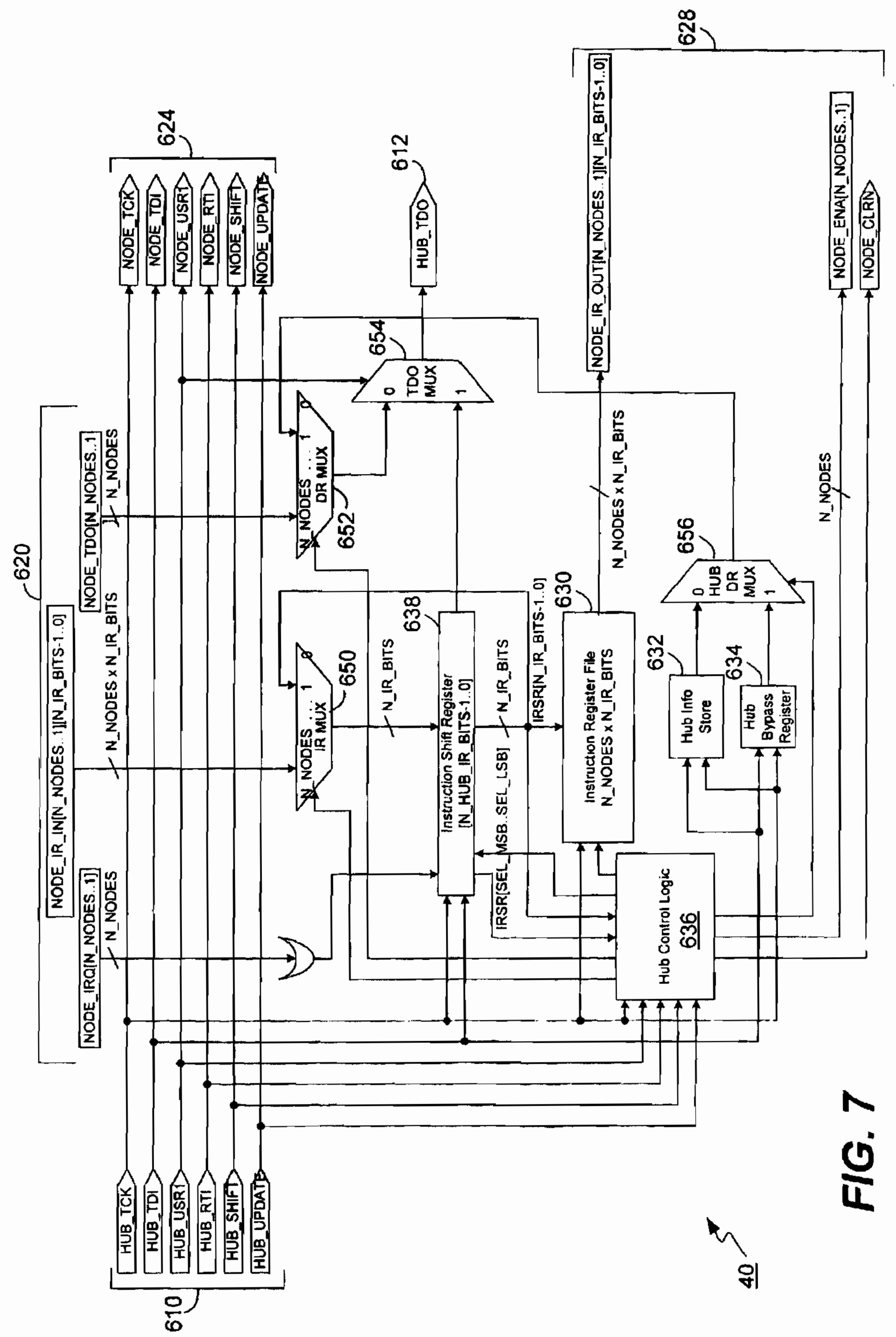

HUB_TCK
HUB_TDI
HUB_USR1
HUB_RTI
HUB_SHIFT
HUB_UPDATE
610
NODE_IRQ[N_NODES..1]
N_NODES
NODE_IR_IN[N_NODES..1][N_IR_BITS-1..0]
N_NODES x N_IR_BITS
NODE_TDO[N_NODES..1]
N_NODES
620
NODE_TCK
NODE_TDI
NODE_USR1
NODE_RTI
NODE_SHIFT
NODE_UPDATE
624
HUB_TDO
612
N_NODES ... 1 0 IR MUX
650
N_NODES ... 1 0 DR MUX
652
0 TDO MUX 1
654
N_IR_BITS
Instruction Shift Register [N_HUB_IR_BITS-1..0]
638
IRSR[SEL_MSB..SEL_LSB]
N_IR_BITS
IRSR[N_IR_BITS-1..0]
Instruction Register File N_NODES x N_IR_BITS
630
Hub Control Logic
636
Hub Info Store
632
Hub Bypass Register
634
0 HUB DR MUX 1
656
N_NODES x N_IR_BITS
NODE_IR_OUT[N_NODES..1][N_IR_BITS-1..0]
N_NODES
NODE_ENA[N_NODES..1]
NODE_CLRN
628
40

FIG. 7

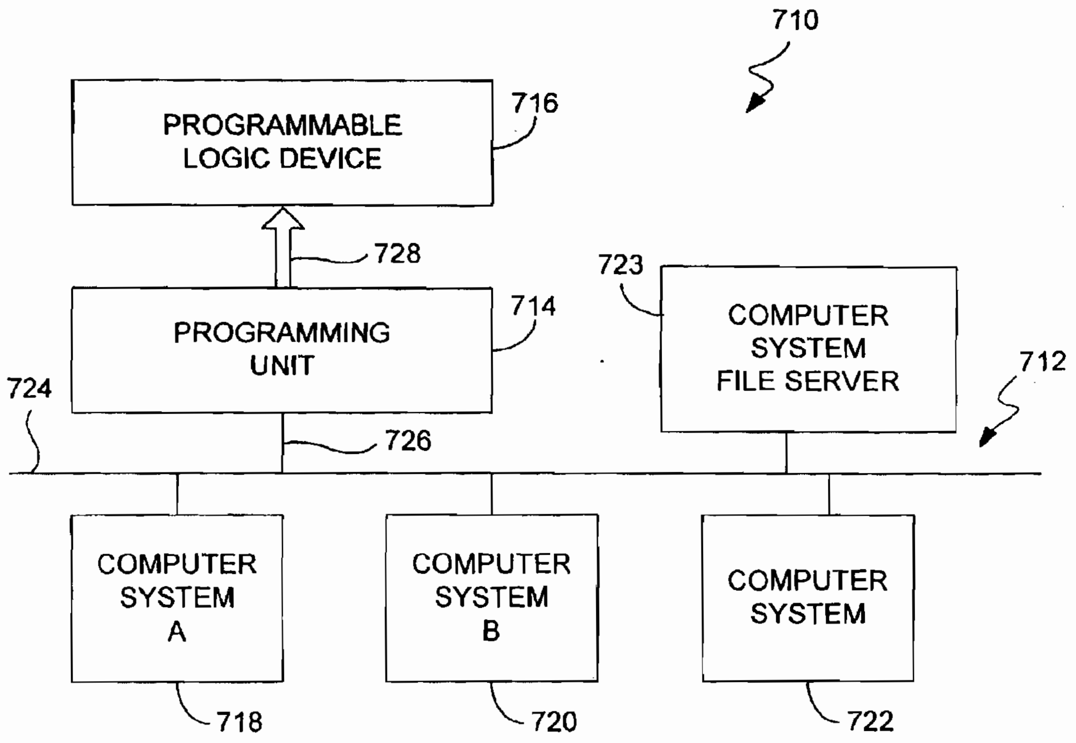

710
PROGRAMMABLE LOGIC DEVICE
716
728
723
PROGRAMMING UNIT
714
COMPUTER SYSTEM FILE SERVER
712
724
726
COMPUTER SYSTEM A
COMPUTER SYSTEM B
COMPUTER SYSTEM
718
720
722

FIG. 8

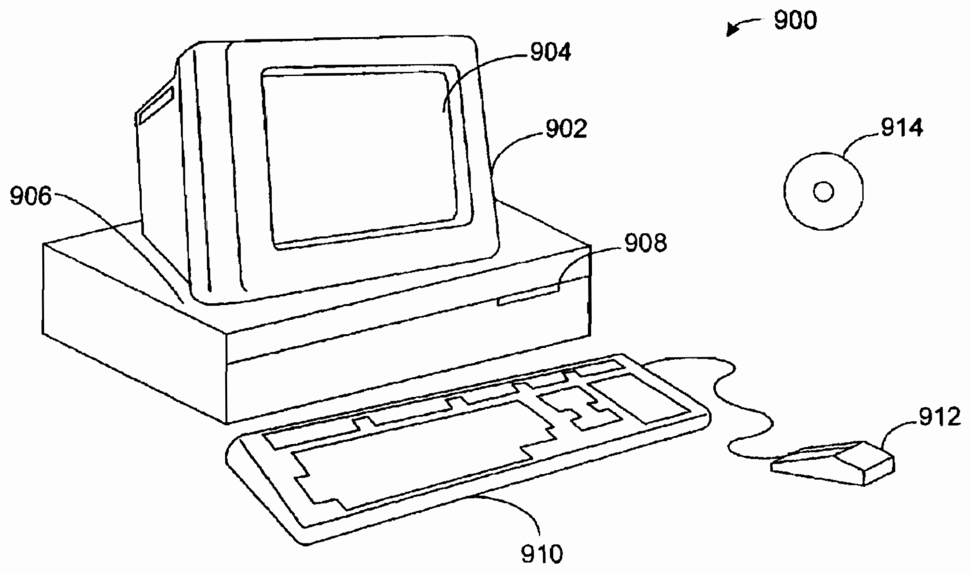

900
904
902
914
906
908
912
910

FIG. 9A

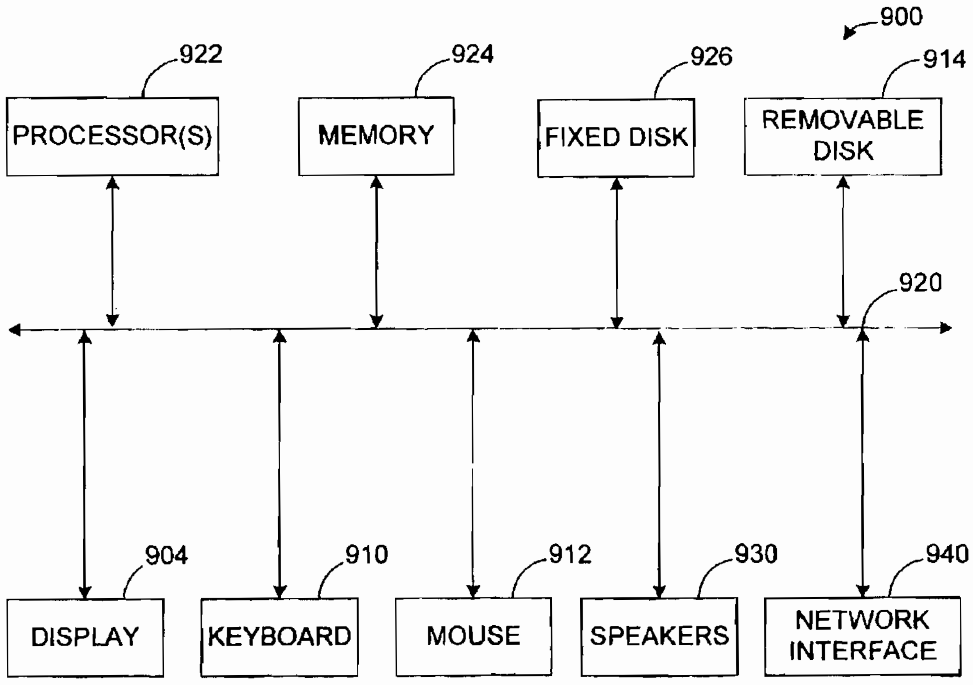

900
922
924
926
914
PROCESSOR(S)
MEMORY
FIXED DISK
REMOVABLE DISK
920
904
910
912
930
940
DISPLAY
KEYBOARD
MOUSE
SPEAKERS
NETWORK INTERFACE

FIG. 9B

PLD DEBUGGING HUB

This application is related to U.S. patent application Ser. Nos. 10/351,017 and 10/629,508, and to U.S. Pat. Nos. 6,182,247, 6,247,147, 6,286,114, 6,389,558 and 6,460,148 and 6,704,889 and 6,754,862, which are all hereby incorporated by reference.

FIELD OF THE INVENTION

The present invention relates generally to analysis of a hardware device in connection with a computer system. More specifically, the present invention relates to control of multiple debugging tools within a programmable logic device.

BACKGROUND OF THE INVENTION

In the field of electronics various electronic design automation (EDA) tools are useful for automating the process by which integrated circuits, multi-chip modules, boards, etc., are designed and manufactured. In particular, electronic design automation tools are useful in the design of standard integrated circuits, custom integrated circuits (e.g., ASICs), and in the design of custom configurations for programmable integrated circuits. Integrated circuits that may be programmable by a customer to produce a custom design for that customer include programmable logic devices (PLDs). Programmable logic devices refer to any integrated circuit that may be programmed to perform a desired function and include programmable logic arrays (PLAs), programmable array logic (PAL), field programmable gate arrays (FPGA), complex programmable logic devices (CPLDs), and a wide variety of other logic and memory devices that may be programmed. Often, such PLDs are designed and programmed by a design engineer using an electronic design automation tool that takes the form of a software package.

In the course of generating a design for a PLD, programming the PLD and checking its functionality on the circuit board or in the system for which it is intended, it is important to be able to debug the PLD because a design is not always perfect the first time. Before a PLD is actually programmed with an electronic design, a simulation and/or timing analysis may be used to debug the electronic design. Once the PLD has been programmed within a working system, however, it is also important to be able to debug the PLD in this real-world environment.

And although a simulation may be used to debug many aspects of a PLD, it is nearly impossible to generate a simulation that will accurately exercise all of the features of the PLD on an actual circuit board operating in a complex system. For example, a simulation may not be able to provide timing characteristics that are similar to those that will actually be experienced by the PLD in a running system; e.g., simulation timing signals may be closer or farther apart than what a PLD will actually experience in a real system.

In addition to the difficulties in generating a comprehensive simulation, circuit board variables such as temperature changes, capacitance, noise, and other factors may cause intermittent failures in a PLD that are only evident when the PLD is operating within a working system. Still further, it can be difficult to generate sufficiently varied test vectors to stress the PLD design to the point where most bugs are likely to be observed. For example, a PLD malfunction can result when the PLD is presented with stimuli that the designer did not expect, and therefore did not take into account during the design and simulation of the PLD. Such malfunctions are

difficult to anticipate and must be debugged in the context of the complete system. Thus, simulation of an electronic design is useful, but usually cannot debug a PLD completely.

One approach to debugging a hardware device within a working system is to use a separate piece of hardware equipment called a logic analyzer to analyze signals present on the pins of a hardware device. Typically, a number of probe wires are connected manually from the logic analyzer to pins of interest on the hardware device in order to monitor signals on those pins. The logic analyzer captures and stores these signals for later viewing and debugging.

As an external logic analyzer may not always be optimal, embedding a logic analyzer within the hardware device is another technique used. For example, U.S. Pat. No. 6,182,247 entitled "Embedded Logic Analyzer for a Programmable Logic Device" discloses such a technique, and U.S. Pat. Nos. 6,286,114 and 6,247,147 disclose enhancements. In addition, viewing internal nodes in a device may be performed as disclosed in U.S. Pat. Ser. No. 6,754,862. Embedding a logic analyzer into a design is also a technique used in the product "ChipScope ILA" available from Xilinx Inc., of San Jose, California. The product "ChipScope Pro" also available from Xilinx uses logic cores built directly into a PLD to allow a user to access internal signals and nodes for debugging.

As useful as these techniques are in debugging a PLD, there is room for improvement. For example, as described in U.S. Pat. No. 6,286,114, a user controls a single embedded logic analyzer through a JTAG port. While such a technique is extremely useful, in many situations it would be desirable to have more than one internal debugging tool have access to the JTAG port, while still maintaining the benefits of a direct interface. In other words, it would be desirable for the user to be able to communicate with, and control, any number of internal logic analyzers, other debugging tools, or other applications through the JTAG port or a suitable serial interface.

For example, a PLD may use two different clock domains (or more) such as a 100 MHz and a 50 MHz clock. With two different clock speeds, a single embedded logic analyzer might not be able to capture debugging data from within the different clock domains. It would be useful to have two or more logic analyzers, each running at a different clock speed and still communicating to the user via a single, serial interface. The user may also wish to capture data from within different parts of the PLD using two or more different trigger conditions. Again, having more than one logic analyzer would be very useful.

The ChipScope product available from Xilinx, Inc. does provide the ability to have multiple logic analyzers within a PLD. It is believed, though, that these logic analyzers must be placed in series within the PLD which has disadvantages. For example, a user or software application desiring to access one of the logic analyzers using the ChipScope product needs to know about all of the internal logic analyzers and where the particular analyzer sits in the series chain. Requiring a user or software tool to be aware of all internal debugging tools and to coordinate amongst them can be confusing and inefficient.

It would be desirable to allow the user of an EDA tool to communicate with, and control, any number of embedded logic analyzers, debugging tools, or other internal applications that are within a PLD. Further, it would be desirable for the user to be able to control such a tool irrespective of any other internal tool, and to be able to do so via any single JTAG port or other serial interface.

SUMMARY OF THE INVENTION

To achieve the foregoing, and in accordance with the purpose of the present invention, a PLD debugging hub is disclosed that allows any number of client modules embedded within a PLD to communicate to an external computer using a serial interface.

The present invention allows user logic present within a PLD to be debugged by way of the hub. The PLD includes a serial interface that allows communication with a host computer. Within the PLD may be any number of client modules that provide instrumentation for the PLD. Each client module has connections with the user logic that allows the instrumentation to work with the user logic. The hub communicates with each client module over a hub/node signal interface, and communicates with the serial interface over a user signal interface. The hub may route instructions from the host computer to any client module via the serial interface.

In one sense, the hub disclosed functions as a multiplexor, allowing any number of client modules (or "nodes") to communicate externally though a serial interface of the PLD as if each node were the only node interacting with user logic. In this way, it is transparent that other nodes may also be present inside the PLD and control is simpler.

The hub described herein exists between a serial interface and user logic and provides a mechanism for sharing communication over a JTAG port (in one embodiment) amongst multiple, heterogeneous client modules. These client modules (such as logic analyzers, debugging tools or other) may operate independently and without knowledge of each of the other modules. These client modules include but are not limited to: a logic analyzer for capturing debugging data; a fault injector for forcing internal nodes to certain values for debugging; a debugging system controller used for controlling a debugging system within a microprocessor; and a signal source (also called a "programmable ROM") for use as "soft" constants in DSP or other applications that make use of fixed values. Other types of client modules are also possible.

Unlike prior art techniques that might use multiple internal logic analyzers in a series within a PLD, the present invention does not require a user or software tool to know about such other modules within the PLD. The existence of other modules is transparent when a user is communicating with a single module. Any EDA tool communicating with and controlling a particular module need not be aware of, and need not coordinate with, any other internal client module. The advantage is that control is simplified by providing and maintaining a uniform client module interface, while allowing the flexibility and scalability of adding other, possibly unrelated, client modules to the hub. This allows the EDA tool to be designed to interface to its client modules in such a way that the communication appears to be exclusive to a module, regardless of the actual configuration of the hub or of the presence of other modules.

BRIEF DESCRIPTION OF THE DRAWINGS

The invention, together with further advantages thereof, may best be understood by reference to the following description taken in conjunction with the accompanying drawings in which:

FIG. 1 is a block diagram of a programmable logic device (PLD) that embodies the present invention.

FIG. 2 illustrates the functional relationship between external JTAG signals and the user debug signal interface.

FIG. 3 illustrates an alternative embodiment in which the hub communicates outside the PLD using a serial interface.

FIG. 4 illustrates examples of possible client modules.

FIG. 5 is a flow diagram illustrating one embodiment in which a PLD is programmed for debugging.

FIG. 6 is a flow diagram illustrating one embodiment in which a PLD is debugged.

FIG. 7 is a block diagram of one embodiment of the hub.

FIG. 8 is a block diagram of an embodiment of a programmable logic development system.

FIGS. 9A and 9B illustrate a computer system suitable for implementing embodiments of the present invention.

DETAILED DESCRIPTION OF THE INVENTION

FIG. 1 is a block diagram of a programmable logic device (PLD) **10** that embodies the present invention. Included are user logic **20**, any number of client modules **30–36**, a hub **40**, interfaces **60** and **70**, and a JTAG interface **50** and **52**. User logic **20** is any electronic design created by a user that is programmed into a PLD; techniques for designing user logic and for programming a PLD are well known in the art. Client modules **30–36** (or "nodes") may be any module programmed into the PLD. In general, a module is a specific piece of instrumentation used to analyze, control or debug the user logic **20**. As mentioned above, a module may be an embedded logic analyzer, a fault injector, a debugging system controller, a signal source, or other client instrumentation. FIG. **4** illustrates examples of possible client modules (or "nodes"). The core/node connections **22–28** are the responsibility of each module, i.e., the particular connections and how they are implemented will be specific to each module. Implementation of such connections is known in the art, and is also described in the above-referenced prior art in the Background.

PLD **10** highlights the interfaces between hub **40**, the JTAG State Machine (JSM) **52**, the nodes, the user logic **20**, and the external JTAG signals **50** (TCK, TMS, TDI and TDO). The hub, nodes, user logic and their interconnections are preferably soft (i.e., realized in core logic). The four JTAG pins, their connection to the JSM, and JSM itself are preferably hard (i.e., dedicated hardware resources provided by the PLD). Alternatively, the hub and nodes may be a dedicated hardware resource of a PLD, in which case a particular PLD would be designed specifically to provide certain nodes. Or, the JSM may be implemented in core logic, providing more flexibility for the invention to be implemented on any PLD, and not necessarily on a PLD with a dedicated JSM.

The JTAG port includes JSM **52** and pins **50**. A JTAG (Joint Test Action Group) port is implemented under the IEEE 1149.1 standard and is known to those of skill in the art. In this embodiment, the JTAG port includes signals TCLK, TMS, TDI and TDO. Signal TCLK is a clock signal that controls the rate of serial data in and out of the JTAG port. Signal TMS is a mode select signal used to direct traversal of the sixteen states of the JSM. Signals TDI and TDO are serial data in and serial data out, respectfully. JSM **52** is a standard part of the JTAG port and is preferably hard logic. It is also referred to as the test action port (TAP) controller.

Typically, a JTAG port is used either to program a PLD or to assist with testing a circuit board on which PLDs are located. Advantageously, it is realized that a JTAG port has traditionally been unused during the design and debugging of a particular PLD. Thus, it is further realized that a JTAG port on a PLD is under utilized and may be used during debugging of a PLD as a means of communicating with and controlling any number of internal client modules.

User Debug Signal Interface

The signal interface between JSM **52** and hub **40** is termed a user debug signal interface **70**. In one embodiment, it is a hard interface that includes 7 signals. These 7 signals are listed below in Table 1. The hub signal port column shows the corresponding connection at hub **40**.

The user debug signals are provided by the JSM and may be connected to core routing resources. The user debug signals are active when either the USER0 or USER1 JTAG instruction is the active instruction in the JSM. This condition is referred to as user debug mode (UDM). Unlike other JTAG instructions that use dedicated hardware resources to realize their target data registers, the target data register for these two instructions is realized in core logic. The user debug signals are used to control the communication to these registers.

The user debug signals are inactive when the instruction in the JSM is not USER0 or USER0 so that the content of their target data register is maintained while other operations are performed on the JTAG port.

TABLE 1

JSM Signal Port	Hub Signal Port	Description
CLOCK_U	→ HUB_TCK	A gated TCK, active when the JSM is in states CDR or SDR
TDI_U	→ HUB_TDI	Directly connected to TDI and always available
RUNIDLE_U	→ HUB_RTI	Indicates that the JSM is in the RTI state
SHIFT_U*	→ HUB_SHIFT	Indicates that the JSM is in the SDR state
UPDATE_U	→ HUB_UPDATE	Indicates that the JSM is in the UDR state
USR1_U	→ HUB_USR1	Indicates that current instruction in the JSM is the USER1 instruction
TDO_U	← HUB_TDO	Connected to TDO when in UDM and the JSM is in state SDR

FIG. **2** illustrates the functional relationship **100** between external JTAG signals **50** and the user debug signal interface **70**. Shown are signals **102** that are signals present on the JSM signal port as described in Table 1. In this example, PLD **10** has a hub with three nodes and a maximum node instruction length of **10**. The first four timing signals in the FIG. are of JTAG signals **50**, while the next six timing signals are those from the JSM signal port (with the exception of signal TDO_U which in an input to the port). FIG. **2** illustrates one example of how particular combinations of JTAG signals **50** are used to produce outputs over user debug signal interface **70**. Of course, more signals may be added to the user debug signal interface, or there may be fewer. For instance, the TCK and TMS signals could be used, in addition to or in lieu of some signals in the user debug signal interface defined in Table 1, to provide more control and resolution into the current state of the JSM. Also, the signals may be encoded differently, outputs may be triggered on a falling edge instead of on a leading edge and vice-versa, etc.

Looking at FIG. **2** from left to right, the JSM moves from RTI (not shown) to SIR, where the USER1 instruction (0000001110) is shifted in (LSB to MSB). Upon the falling edge of TCK when the JSM is in UIR, the USR1_U signal goes high, indicating the USER1 instruction is now the active instruction in the JSM (i.e., the JSM is in user debug mode). Consequently, SHIFT_U goes low. Next, the JSM moves to SDR, where 12 bits of zero are shifted in. Since USER1 is the active instruction in the JSM, this corresponds to an instruction load for hub **40** with the HUB_INFO instruction (irsr[11 . . . 10]='00' and irsr[2 . . . 0]='000'). The JSM then moves to SIR, where the USER0 instruction (0000001100) is shifted in, and then to SDR, where the first 4 bits held in hub **40** Info Store **632** are shifted out.

Alternative Serial Interface

FIG. **3** illustrates an alternative embodiment **200** in which hub **40** communicates outside the PLD **10** using a serial interface. The serial interface shown is by way of example, and other types of serial interfaces may be used. In this example, communication takes place over four pins of the PLD; of course, any number of pins may be used to perform a similar function. Pins **202** and **204** transmit respectively a data in signal and a data out signal to and from the PLD. Pin **206** is a clock signal provided by the external computer which is used to synchronize the serial transmission of commands, data and other information from the external computer to the PLD, and from the PLD to the external computer.

Mode select pin **208** is used to transmit commands, modes and other control information from the external computer to hub **40**. Mode select **208** may use more than one pin and may also be used to receive status or other information from the hub. Mode select may also transmit identifying information for a particular client module from the external computer to the PLD. Pin **208** may be connected to a separate mode input of hub **40** or to any of the signals listed in Table 1 as appropriate. In this fashion, the hub communicates with an external computer using a serial interface that is not necessarily a JTAG port.

Hub/Node Signal Interface

The signal interface between hub **40** and any node **30**–**36** is termed the hub/node signal interface. This is preferably a soft interface that includes 5 buses and 7 signals. The hub/node signals are connected using core routing resources. Such connections may easily be made by one of skill in the art. Table 2 below shows the hub/node signal interface with respect to a particular node. In other words, although hub **40** may be connected to any number of nodes, the second column of Table 2 shows only those connections on a single node. Should there be more than one node, each node would have the connections shown in the second column. In an embodiment where Hub/Node connections are made automatically by a netlist builder tool (e.g., an EDA tool), it is preferable that that connected nodes use the bus/signal name definitions shown in Table 2. By way of example, such a tool may be the "Quartus" product available from Altera Corporation, or other netlist builder tool.

TABLE 2

Hub Bus/Signal Port	Node i Bus/Signal Port	Description
NODE_TCK	→ TCK	Node clock (common to all nodes)
NODE_TDI	→ TDI	Node data in (common to all nodes)
NODE_RTI	→ RTI	Indicates that the JSM is in the RTI state (common to all nodes)
NODE_SHIFT	→ SHIFT	Indicates that the JSM is in the SDR state (common to all nodes)
NODE_UPDATE	→ UPDATE	Indicates that the JSM is in the UDR state (common to all nodes)
NODE_USR1	→ USR1	Indicates that current instruction in the JSM is the USER1 instruction (common to all nodes)
NODE_CLRN	→ CLRN	Asynchronous clear (common to all nodes)
NODE_ENA[i]	→ ENA	Indicates that the current instruction in hub 40 is for node i
NODE_IR_OUT[i] [N_NODE_IR_BITS(i)-1..0]	→ IR_IN[N_NODE_IR_BITS(i)-1..0]	Node i IR
NODE_TDO[i]	← TDO	Node i data out
NODE_IRQ[i]	← IRQ	Node i interrupt
NODE_IR_IN[i] [N_NODE_IR_BITS(i)-1..0]	← IR_OUT[N_NODE_IR_BITS(i)-1..0]	Node i IR capture port

Details on the connections shown in Table 2 are as follows. The variable N_NODE_IR_BITS(i) is the number of instruction register bits required by a node i. The signals NODE_TCK, NODE_TDI, NODE_RTI, NODE_SHIFT, NODE_UPDATE and NODE_USR1 of the hub port for the nodes are directly connected to the signals HUB_TCK, HUB_TDI, HUB_RTI, HUB_SHIFT, HUB_UPDATE and HUB_USR1 of the hub port for the JSM, respectively.

The NODE_CLRN signal is an asynchronous, active low clear signal that is activated when the JSM is in RTI after the HUB_RESET instruction becomes the active Hub instruction. Since hub **40** is also reset by this signal, the HUB_INFO instruction becomes the active Hub instruction.

The NODE_ENA[i] bus is a one-hot bus that is used to inform a node that the current hub instruction is for that node, e.g. if $NODE_{13}$ ENA[3] is 1, then an instruction for node **3** is the current instruction in the hub's instruction register. This means that when NODE_SHIFT is 1, the associated target register for their instruction is part of the JTAG TDI→TDO scan chain. Moreover, this places the burden on nodes to provide a path between TDI and TDO. Preferably, there is no discontinuity between TDI and TDO when NODE_SHIFT is 1. For the HUB_INFO instruction, a 4-bit shift register is used between TDI and TDO. For other hub instruction patterns, hub bypass register **634** is between TDI and TDO.

Hub **40** provides the instruction register resource for all nodes, and nodes obtain their instruction from their respective NODE_IR_OUT[i] port of the Hub. Hub **40** stores the instruction for each node in instruction register file **630**. Node TDOs are fed to their corresponding NODE_TDO[i] input port of the Hub.

The NODE_IRQ[i] port is provided so that nodes may indicate that they need attention, i.e., a node has a result or stored information that should be communicated externally back to the user or EDA tool. For example, a node that is a logic analyzer may have captured data that needs to be sent back to a host computer to aid in debugging the PLD. In one embodiment, this interrupt feature is implemented as follows. All of the NODE_IRQ[i] inputs are OR'ed together, and made available on the MSB of USER1 data register scans (i.e., UDM instruction loads).

This single bit interrupt flag indicates the existence of a service request on one or more Nodes. Due to the nature of the shared JTAG user debugging access that hub **40** provides, a node should keep its IRQ signal high until the node is serviced. The host agent (such as an EDA tool running on a host computer) controlling the communication with the nodes polls each node it controls to see which (if any) nodes need to be serviced. All nodes share the same interrupt level, so the host agent should establish a pecking order if multiple nodes need to be serviced simultaneously. Alternatively, a rigid interrupt level may be established in which nodes are serviced in a particular order. When the host agent decides to service a particular node, the host agent executes the node's interrupt service routine. This series of operations is specific to a node, and is executed by issuing instructions and/or performing data exchange operations on the node. Once this routine is complete, it is either up to the host agent to direct the node to clear its IRQ signal, or the node's logic to automatically acknowledge that its interrupt has been serviced and clear the IRQ signal without further intervention from the host agent.

One method of communication from a node to the outside world that avoids the overhead of accessing the target data register of the node's instruction utilizes the NODE_IR_IN**[i]** bus and the hub's instruction register (IR) capture value. A given node **i** can use the NODE_IR_IN[i] bus of hub **40** to provide the IR capture value during UDM instruction loads when the current hub instruction is for node i. In this way, information may be transferred without accessing the target data register of the instruction currently being applied in the UDM instruction load sequence. It also allows for node i with an instruction that targets a read-only data register to save PLD resources and use a single register as the instruction's target data register, while providing the read-only information in the IR capture value assuming that the read-only data length is of equal or lesser value than the IR length of node i. The HUB_FORCE_IR_CAPTURE instruction may be used to force the IR capture value to be from a node other than the one targeted by the current hub instruction. This feature is very useful in that it may not be known which instruction currently resides in the hub (i.e., it may not be possible to ensure that an instruction for a particular node was the last one issued), and the IR capture value for a particular node is required. Issuing HUB_FORCE_IR_CAPTURE prior to the issuance of an instruction for node i will guarantee that the IR capture value is from node i. The IR capture value is undefined when hub **40** is in broadcast mode.

Examplary Flow Diagrams

FIG. **5** is a flow diagram illustrating one embodiment in which a PLD is debugged. Of course, other similar design methodologies may be used, including those referenced in the Background section. A user first develops a design for a PLD using an EDA tool and then compiles the design. The design is then programmed into a PLD (such as PLD **10** with the capability to implement the present invention. The user then debugs the PLD, and, assuming that bugs are found in the design, proceeds as follows.

The user returns to the design and instructs the EDA tool to add a hub **40** and signal interfaces as described herein. The user then instructs the EDA tool to add the instrumentation needed (e.g., logic analyzers, fault injectors, etc.). Alternatively, the PLD may have been preprogrammed for this eventuality and already includes this logic. The commands may be given via a graphical user interface (GUI), by directly adding the required functionality to the design, or in other similar ways. Advantageously, any number and type of instrumentation may be added, constrained only by the size of the PLD. Techniques for adding a particular instrumentation will vary by the type, and are known to those of skill in the art. For each instrumentation, the user supplies a manufacturer identifier, a node identifier, a node version number, and a node instance number.

The user next performs a recompile to include all of the added instrumentation, hub, signal interfaces, etc. During the recompile, the EDA tool (or compiler) assigns each node a selection identifier to aid in sending instructions and data to a node, as well as to aid in receiving information from a node. The selection identifier is a unique identifier for a particular node in the PLD, and is preferably derived from a combination of the identifiers listed above, although the selection identifier may be derived from other information as well. The new design is then programmed onto a PLD and the user may debug once again using any of the instrumentation added.

FIG. **6** is a flow diagram illustrating one embodiment in which a PLD is debugged. Once the instrumentation and hub have been added (for example, as shown in FIG. **5**), the user is ready to begin using the instrumentation. To identify each node, and to keep operation of other nodes transparent for a chosen node, the EDA tool and hub use the selection identifier to direct instructions to a node and to poll a node for information. Preferably, the selection identifier precedes an instruction for a node, although this could be reversed. In a first step, an instruction is issued from the EDA tool to arm, enable, or otherwise turn on the embedded instrumentation. Each may be turned on separately, or all may be turned on together.

To provide an instruction to a node (e.g., run, stop, trigger condition, control command, etc.), the EDA tool provides the instruction to the hub via the JTAG interface preceded by the node's selection identifier. To receive information from a node, polling or interrupts may be used. Typically, the outside host agent (EDA tool or other software) periodically polls the JTAG interface for a signal that the node is ready with information. Alternatively, a node may send back an interrupt in an instruction scan. The interrupt may be a set flag, a particular instruction, etc. Once the host agent has detected that a node is ready with information (by polling, interrupt, etc.), the host issues a read instruction preceded by the node's selection identifier in order to receive the information, or take other appropriate action.

Hub Implementation Example

FIG. **7** is a block diagram of one embodiment of hub **40**. Hub **40** serves to allow communication between any number of nodes **30**–**36** and a JTAG port (or any other suitable serial interface) so that a node may interact with user logic **20** as a user desires. Typically, a node may be a logic analyzer and hub **40** facilitates control of that logic analyzer, for example. As previously described, signals **610** are received as input from JSM **52**, and signal **612** is output to the JSM. Buses **620** are input from any number of nodes, and signals and buses **624** and **628** are output to any nodes that are present.

The data registers associated with the JTAG USER**0** and USER**1** instructions are user-defined. To provide instructions and information to the hub or to a node, the operation of the hub mimics the instruction/data register (IR/DR) paradigm defined by the JTAG standard. This operation is accomplished by designating the user-defined DR for the USER**1** instruction as the hub's instruction register, or instruction shift register (IRSR). Correspondingly, the USER**0** instruction targets the hub's data register. All nodes also follow this paradigm. Therefore, to issue an instruction to either hub **40** or a Node, the USER**1** instruction is the active instruction in the JSM, and the instruction is shifted in when the JSM is in the SDR state. Similarly, to shift in associated target register data, the USER**0** instruction is the active instruction in the JSM, and the associated target register data is shifted in when the JSM is in the SDR state.

Hub **40** includes an instruction register file **630**, a hub information store **632**, a hub bypass register **634**, hub control logic **636**, and an instruction shift register **638**. Also included are multiplexors **650**–**656**. As will be appreciated by one of skill in the art, hub **40** may implemented in other ways, yet still provide the same functionality.

Instruction register file (IRF) **630** consists of the instruction registers for all nodes. In other words, IRF **630** stores instructions for all nodes in hub **40** before the instruction is sent out to each node. Advantageously, each node need not store any instructions, and need not look at all instructions provided to hub **40**, thus providing greater efficiency. Instruction shift register (IRSR) **638** receives instruction information from the JSM via HUB_TDI serially and transfers it into the proper destination in the IRF **630** as directed by the hub control logic (HCL) **636**. In addition, IRSR **638** receives information from a node (i.e., the IR capture value) and transfers it serially back to the JSM via HUB_TDO **612**.

Hub information store **632** is a repository of hub configuration and node identification information. For example, store **632** keeps a record of: a manufacturer identifier, indicating from which manufacturer a node originates; a node identifier, indicating the type of instrumentation that the node embodies; a node version number, indicating the version of that particular type of instrumentation; and a node instance number, which uniquely identifies a node. Further details are provided below in Table 6 as an example of the type of information that is stored to ensure proper operation. Preferably, this information is compiled into the hub when first compiled. Other information may be included in store **632** as deemed necessary. For instance, PLD resource usage information for nodes may be kept in store **632**, or other important information necessary for proper use and control of a node.

Hub bypass register **634** is used to maintain JTAG continuity when there is no target register for the hub, or if a targeted node is outside the valid node selection space. Register **634** provides a default path for any instruction without a valid target register, or an invalid data register. Hub control logic (HCL) **636** generates the necessary control signals used throughout the Hub. The User Debug Signal Interface and the value in instruction shift register **638** (ISR) provide the input stimulus to the HCL. The outputs from the HCL control the multiplexors used to steer data to its proper destination, and serve as register enable controls. HCL **636** also has the function of maintaining JTAG continuity if an "out-of-bounds" instruction is issued, e.g., if an instruction targets a node outside the valid node selection space the HCL will maintain JTAG continuity by placing hub bypass register between HUB_TDI and HUB_TDO. Alternatively, HCL may be implemented as two logical boxes instead of the single one shown. For example, the HCL may be broken into those signals and logic that control the hub, and those that are used for controlling output.

The following parameters are used to specify the hub, and are provided by the netlist builder tool.

TABLE 3

Parameter	Definition
N_NODES	The number of nodes connected to the hub
N_IR_BITS	MAX(N_NODE_IR_BITS(i))
NODE_INFO(i)	A 32-bit value (described below)

The hub also makes use of the following constant definitions.

TABLE 4

Constant	Definition
N_SEL_BITS	CEIL(LOG2(N_NODES + 1)
N_HUB_IR_BITS	N_SEL_BITS + N_IR_BITS
SEL_MSB	N_HUB_IR_BITS - 1
SEL_LSB	N_IR_BITS

The length of the hub's instruction register (IRSR) is 'N_HUB_IR_BITS' bits, which is the sum of N_IR_BITS (i.e., MAX(N_NODE_IR_BITS(i)), the maximum of the node and the minimum hub IR lengths) and N_SEL_BITS (i.e., the number of bits required to encode the number of nodes plus the hub). This encoded value (SELect) allows for the hub and all nodes to have non-conflicting instruction codes. By definition, the minimum hub IR length N_NODE_IR_BITS(Hub)=N_SEL_BITS+3, SEL(Hub)=0 always, and SEL(Node(i))=i. Table 5 below shows the instructions supported by the hub.

TABLE 5

Instruction	Value	Description
HUB_INFO	0	Provides information about the hub and all of the nodes
HUB_START_BROADCAST	1	Delays instruction updates to nodes until HUB_END_BROADCAST is issued
HUB_END_BROADCAST	2	Updates node instructions
HUB_FORCE_IR_CAPTURE	3	Forces the instruction capture of the next instruction load to come from the specified node
HUB_RESET	7	Asserts NODE_CLRN while JSM is in RTI

When the HUB_INFO instruction is issued, the data in hub information store **632** is shifted out 4 bits at a time, i.e., multiple cycles through the data register (DR) leg of the JSM are required to retrieve all the data. Each nibble is loaded on the rising edge of HUB_TCK when the JSM is in the CDR state. The information held in hub information store **632** is packed into lookup table (LUT) CRAM cells to reduce resource usage. The data is shifted out LSB to MSB as shown in Table 6.

TABLE 6

DWORD\BIT	31 27	26 19	18 8	7 0
0	HUB_VERSION	N_NODES	MFG_ID	N_IR_BITS
1	NODE$_1$ VERSION	NODE$_1$ ID	NODE$_1$ MFG_ID	NODE$_1$ INSTANCE
	.	.	.	.
	.	.	.	.
	.	.	.	.
N	NODE$_N$ VERSION	NODE$_N$ ID	NODE$_N$ MFG_ID	NODE$_N$ INSTANCE

HUB_VERSION and MFG_ID are embedded in the source logic of the hub. MFG_ID is a manufacturer's identification number assigned by an entity authorized to maintain unique identifiers for this invention. N_NODES is a parameter provided by the netlist builder tool, and N_HUB_IR_BITS is a constant computed as defined above. The NODE_INFO(i) parameters are concatenated by the netlist builder tool and are passed on as one long parameter to the hub. NODE ID, assigned by the manufacturer of the client module, identifies the type of client module functionality. For example, a NODE ID of 0 could represent a logic analyzer, a NODE ID of 1 could represent a fault injector, etc. NODE VERSION, also assigned by the manufacturer of the client module, represents the version of this particular type of client module. NODE INSTANCE, assigned by an FDA tool inserting the hub and client modules into the PLD design, identifies the instance of a particular NODE ID in the PLD. For example, if there are two logic analyzers in the same PLD from the same manufacturer and each is the same version, NODE INSTANCE distinguishes between the two.

The HUB_START_BROADCAST instruction is used to delay instruction updates to Nodes. This provides the ability to issue instructions simultaneously to the all of the Nodes.

An application of this feature would be to simultaneously arm multiple logic analyzers within the PLD. The HUB_END_BROADCAST instruction updates all NODE_IR_OUT[i] buses.

For nodes that use the instruction register (IR) capture value, the last instruction issued may have been for a different node, and the instruction register capture value will be for that other Node. The HUB_FORCE_IR_CAPTURE instruction can be used to force the IR capture from a particular Node. The format of this instruction for node i is shown below.

TABLE 7

N_HUB_IR_BITS-1 N_SEL_BITS+3	N_SEL_BITS+2 3	2 0
0	i	011

Programmable Logic Development System

In the course of developing an electronic design for programming a programmable logic device (PLD), a programmable logic development system is used. As used herein, "electronic design" refers to a design used to program circuit boards and systems including multiple electronic devices and multi-chip modules, as well as integrated circuits. For convenience, the present discussion generally refers to "integrated circuits", or to "PLDs", although the invention is not so limited.

FIG. **8** is a block diagram of an embodiment of a programmable logic development system **710** that includes a computer network **712**, a programming unit **714** and a programmable logic device **716** that is to be programmed. Computer network **712** includes any number of computers connected in a network such as computer system A **718**, computer system B **720**, computer system C **722** and computer system file server **723** all connected together through a network connection **724**. Computer network **712** is connected via a cable **726** to programming unit **714**, which in turn is connected via a programming cable **728** to the PLD **716**. Alternatively, only one computer system could be directly connected to programming unit **714**. Furthermore, computer network **712** need not be connected to programming unit **714** at all times, such as when a design is being developed, but could be connected only when PLD **716** is to be programmed.

Programming unit **714** may be any suitable hardware programming unit that accepts program instructions from computer network **712** in order to program PLD **16**. By way of example, programming unit **714** may include an add-on logic programmer card for a computer, and a master programming unit, such as are available from Altera Corporation of San Jose, California. PLD **716** may be present in a system or in a programming station. In operation, any number of engineers use computer network **712** in order to develop programming instructions using an electronic design automation software tool. Once a design has been developed and entered by the engineers, the design is compiled and verified before being downloaded to the programming unit. The programming unit **714** is then able to use the downloaded design in order to program PLD **716**.

For the purposes of debugging a PLD according to an embodiment of the present invention, any of the computers shown or others may be used by an engineer to compile a design. Furthermore, programming cable **728** may be used to receive data from the PLD, or a separate debugging cable may be used to directly connect a computer with device **716**. Such a programmable logic development system is used to create an electronic design. A user creates a design by specifying and implementing functional blocks.

The above-referenced U.S. patents disclose a design methodology for using a system design specification in order to develop a design with which to program a PLD. It should be appreciated that the present invention may be practiced in the context of a wide variety of design methodologies, and with diverse electronic design automation (EDA) software tools.

Computer System Embodiment

FIGS. **9**A and **9**B illustrate a computer system **900** suitable for implementing embodiments of the present invention. FIG. **9**A shows one possible physical form of the computer system. Of course, the computer system may have many physical forms ranging from an integrated circuit, a printed circuit board and a small handheld device up to a huge super computer. Computer system **900** includes a monitor **902**, a display **904**, a housing **906**, a disk drive **908**, a keyboard **910** and a mouse **912**. Disk **914** is a computer-readable medium used to transfer data to and from computer system **900**.

FIG. **9**B is an example of a block diagram for computer system **900**. Attached to system bus **920** are a wide variety of subsystems. Processor(s) **922** (also referred to as central processing units, or CPUs) are coupled to storage devices including memory **924**. Memory **924** includes random access memory (RAM) and read-only memory (ROM). As is well known in the art, ROM acts to transfer data and instructions uni-directionally to the CPU and RAM is used typically to transfer data and instructions in a bi-directional manner. Both of these types of memories may include any suitable of the computer-readable media described below. A fixed disk **926** is also coupled bi-directionally to CPU **922**; it provides additional data storage capacity and may also include any of the computer-readable media described below. Fixed disk **926** may be used to store programs, data and the like and is typically a secondary storage medium (such as a hard disk) that is slower than primary storage. It will be appreciated that the information retained within fixed disk **926**, may, in appropriate cases, be incorporated in standard fashion as virtual memory in memory **924**. Removable disk **914** may take the form of any of the computer-readable media described below.

CPU **922** is also coupled to a variety of input/output devices such as display **904**, keyboard **910**, mouse **912** and speakers **930**. In general, an input/output device may be any of: video displays, track balls, mice, keyboards, microphones, touch-sensitive displays, transducer card readers, magnetic or paper tape readers, tablets, styluses, voice or handwriting recognizers, biometrics readers, or other computers. CPU **922** optionally may be coupled to another computer or telecommunications network using network interface **940**. With such a network interface, it is contemplated that the CPU might receive information from the network, or might output information to the network in the course of performing the above-described described method steps. Furthermore, method embodiments of the present invention may execute solely upon CPU **922** or may execute over a network such as the Internet in conjunction with a remote CPU that shares a portion of the processing.

In addition, embodiments of the present invention further relate to computer storage products with a computer-readable medium that have computer code thereon for performing various computer-implemented operations. The media and computer code may be those specially designed and constructed for the purposes of the present invention, or they may be of the kind well known and available to those having skill in the computer software arts. Examples of computer-readable media include, but are not limited to: magnetic media such as hard disks, floppy disks, and magnetic tape; optical media such as CD-ROMs and holographic devices; magneto-optical media such as floptical disks; and hardware devices that are specially configured to store and execute program code, such as application-specific integrated circuits (ASICs), programmable logic devices (PLDs) and ROM and RAM devices. Examples of computer code include machine code, such as produced by a compiler, and files containing higher level code that are executed by a computer using an interpreter.

Although the foregoing invention has been described in some detail for purposes of clarity of understanding, it will be apparent that certain changes and modifications may be practiced within the scope of the appended claims. Therefore, the described embodiments should be taken as illustrative and not restrictive, and the invention should not be limited to the details given herein but should be defined by the following claims and their full scope of equivalents.

We claim:

1. A programmable logic device (PLD) arranged to provide instrumentation for user logic, said PLD comprising:
user logic arranged to perform a function designated by a user;
a serial interface in communication with a host computer;
a client module in communication with said user logic that provides instrumentation regarding said user logic; and
a hub in communication with said client module and with said serial interface, said hub being arranged to route instructions from said host computer to said client module via said serial interface, whereby said client module provides said instrumentation for said user logic of said PLD.

2. A PLD as recited in claim **1** wherein said client module is a logic analyzer, a fault injector, a debugging system controller or a signal source.

3. A PLD as recited in claim **2**, said hub being further arranged to route information from said client module to said host computer via said serial interface.

4. A PLD as recited in claim **1** further comprising:
a user signal interface that provides said communication between said hub and said serial interface; and
a hub/node interface that provides said communication between said hub and said client module.

5. A PLD as recited in claim **4** wherein said serial interface is a JTAG port.

6. A PLD as recited in claim **1** wherein said serial interface is a JTAG port.

7. A PLD as recited in claim **1** further comprising:
a plurality of client modules, each in communication with said user logic and each providing instrumentation regarding said user logic,
wherein said hub is in communication with each of said client modules, said hub being arranged to route instructions from said host computer to one of said client modules via said serial interface, whereby said one client module provides instrumentation for said user logic of said PLD.

8. A PLD as recited in claim **7** further comprising:
an instruction register of said hub that stores instructions for each of said client modules, whereby use of said instrumentation for one of said client modules is transparent to the other client modules.

9. A system arranged to debug user logic in a programmable logic device (PLD), said system comprising:
a host computer;
a serial interface;
a PLD in communication with said host computer over said serial interface, said PLD including,
user logic,
a plurality of client modules, each in communication with said user logic, and
a hub in communication with each client module and with said serial interface, said hub being arranged to route instructions from said host computer to said client modules via said serial interface; and
an EDA software tool of said host computer that is arranged to send instructions to said client modules of said PLD using said serial interface and via said hub, whereby said client modules debug said user logic of said PLD.

10. A system as recited in claim **9** wherein one of said client modules is a logic analyzer, a fault injector, a debugging system controller or a signal source.

11. A system as recited in claim **10**, said hub being further arranged to route information from one of said client modules to said host computer via said serial interface.

12. A system as recited in claim **9** wherein said serial interface is a JTAG port.

13. A system as recited in claim **9**, said hub being further arranged to route information from one of said client modules to said host computer via said serial interface, whereby said one client module debugs said user logic of said PLD.

14. A system as recited in claim **9** further comprising:
an instruction register of said hub that stores instructions for each of said client modules, whereby use of an instrumentation for one of said client modules is transparent to the other client modules.

15. A hub of a programmable logic device (PLD) arranged to assist with instrumentation of said PLD, said hub comprising:

control logic;
- a hub interface from said hub to each of a plurality of client modules, each of said client modules providing instrumentation for said PLD;
- a user interface from said hub to a serial interface of said PLD;
- an instruction register that stores instructions for each of said client modules;
- logic gates arranged to route an instruction of said instruction register to one of said client modules based on a selection identifier; and
- logic gates arranged to receive data from one of said client modules and to store said data in said instruction register, whereby said hub assists with instrumentation of said PLD by communicating with said client modules.

16. A hub as recited in claim **15**, said hub being further arranged to route data from said client module to a host computer via said serial interface.

17. A hub as recited in claim **15** wherein said serial interface is a JTAG port.

18. A computer-readable medium for providing instrumentation for user logic of a programmable logic device (PLD), computer code of said computer-readable medium comprising electronic representations of:
- user logic of said PLD;
- a serial interface arranged to link a host computer with said PLD;
- a client module in communication with said user logic that provides instrumentation regarding said user logic; and
- a hub in communication with said client module and with said serial interface, said hub being arranged to route instructions from said host computer to said client module via said serial interface, whereby said client module provides said instrumentation for said user logic of said PLD.

19. A computer-readable medium as recited in claim **18** wherein said client module is a logic analyzer, a fault injector, a debugging system controller or a signal source.

20. A computer-readable medium as recited in claim **19**, said hub being further arranged to route information from said client module to said host computer via said serial interface.

21. A computer-readable medium as recited in claim **18** wherein said serial interface is a JTAG port.

22. A computer-readable medium as recited in claim **18** further comprising electronic representations of:
- a plurality of client modules, each in communication with said user logic and each providing instrumentation regarding said user logic,
- wherein said hub is in communication with each of said client modules, said hub being arranged to route instructions from said host computer to one of said client modules via said serial interface, whereby said one client module provides instrumentation for said user logic of said PLD.

23. A computer-readable medium as recited in claim **18** wherein said medium is associated with an EDA software tool.

* * * * *