(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
4 July 2013 (04.07.2013)

WIPO | PCT

(10) International Publication Number
WO 2013/101336 Al

(51) International Patent Classification:
G01R 31/3183 (2006.01) *G01R 31/3185* (2006.01)

(21) International Application Number:
PCT/US20 12/06 1824

(22) International Filing Date:
25 October 2012 (25.10.2012)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
13/338,581 28 December 201 1 (28. 12.201 1) US

(71) Applicant: **ALCATEL LUCENT** [FR/FR]; 3 Avenue Octave Greard, F-75007 Paris (FR).

(72) Inventors: **PORTOLAN, Michele;** Centre De Villarceaux, Route De Villejust, F-91620 Nozay (FR). **VAN, Treuren, Bradford;** 600-700 Mountain Avenue, Murray Hill, NJ 07974-0636 (US). **GOYAL, Suresh;** 600-700 Mountain Avenue, Murray Hill, NJ 07974-0636 (US).

(74) Agent: **FINSTON, Martin, I.;** Alcatel-Lucent USA Inc., Attention: Docket Administrator - Room 3B-212F, 600-700 Mountain Avenue, Murray Hill, NJ 07974-0636 (US).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available):* AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LT, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available):* ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:

— *with international search report (Art. 21(3))*

— *before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments (Rule 48.2(h))*

— *with information concerning incorporation by reference of missing parts and/or elements (Rule 20.6)*

(54) **Title:** METHOD AND APPARATUS FOR DEFERRED SCHEDULING FOR JTAG SYSTEMS

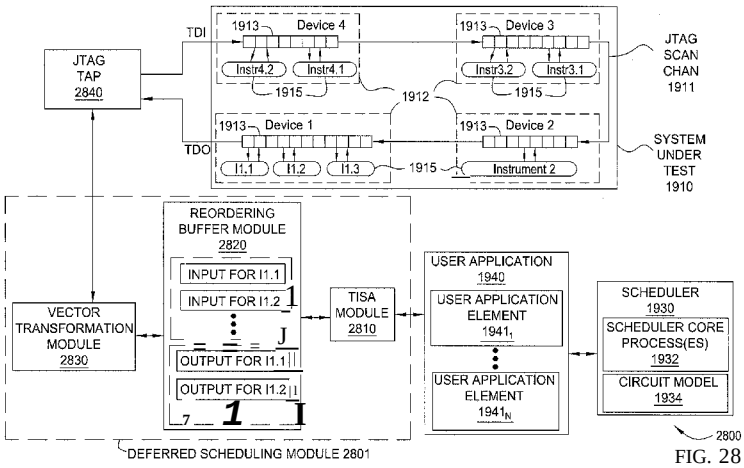

FIG. 28

(57) **Abstract:** A deferred scheduling capability supports deferred scheduling when performing testing via a scan chain of a unit under test. A processing module is configured to receive a plurality of test operations associated with a plurality of segments of a unit under test and to generate therefrom input test data configured to be applied to the unit under test via a Test Access Port (TAP). A reordering buffer module is configured to receive the input test data from the processing element and to buffer the input test data in a manner for reordering the input test data to compose an input test vector for a scan chain of the unit under test. A vector transformation module is configured to receive the input test vector from the reordering buffer module and to apply a vector transformation for the input test vector.

WO 2013/101336 A1

METHOD AND APPARATUS FOR DEFERRED SCHEDULING FOR JTAG SYSTEMS

CROSS-REFERENCE TO RELATED APPLICATIONS

This application is a continuation-in-part of U.S. Patent Application Serial No. 12/495,237, entitled "METHOD AND APPARATUS FOR SYSTEM TESTING USING MULTIPLE INSTRUCTION TYPES" (Attorney Docket No. ALU/1 301 37), U.S. Patent Application Serial No. 12/495,295, entitled "METHOD AND APPARATUS FOR SYSTEM TESTING USING MULTIPLE PROCESSORS" (Attorney Docket No. ALU/1 301 37-2), and U.S. Patent Application Serial No. 12/495,336, entitled "METHOD AND APPARATUS FOR SYSTEM TESTING USING SCAN CHAIN DECOMPOSITION" (Attorney Docket No. ALU/1 301 37-3), each of which was filed on June 30, 2009, and each of which claims the benefit of U.S. Provisional Patent Application Serial No. 61/1 57,41 2, filed on March 4, 2009, entitled TEST INSTRUCTION SET ARCHITECTURE, which applications are hereby incorporated herein by reference in their entirety. This application is related to U.S. Patent Application Serial No. 13/33,8431 , entitled "METHOD AND APPARATUS FOR POSITION-BASED SCHEDULING FOR JTAG SYSTEMS" (Attorney Docket No. 809732-US-CIP), which is hereby incorporated herein by reference in its entirety.

TECHNICAL FIELD

The invention relates generally to JTAG systems and, more specifically but not exclusively, to scheduling of testing in JTAG systems.

BACKGROUND

In Joint Test Action Group (JTAG) applications, access to a System Under Test (SUT) is provided through application of vectors to a scan chain of the SUT. The vectors are serially applied to the standardized Test Access

Port (TAP), which provides an interface to the scan chain of the SUT. The vectors represent the binary value of each bit in the scan chain. A typical test routine may include anywhere from a few vector bits to a large number of vector bits (e.g., hundreds, thousands, or even more). As a result, given that vector sets must be computed and serially applied to the TAP, the size of a vector set is an important consideration in testing for a SUT, especially in embedded applications. Furthermore, vectors are computed for a specific JTAG setup and, thus, need to be modified each time that the JTAG setup changes (an operation that is known as retargeting, which is quite onerous especially when the subsystem to retarget is complext). As a result, dynamic testing (e.g., situations where the vector inputs must be adapted "on the fly" based on the actual outputs) is limited because it requires a continuous retargeting of the entire scan chain and, similarly, use of dynamic topologies (e.g., where the length of the scan chain varies depending on the value of some elements) is limited as concurrent use of instruments is quite difficult because of the computational intesity associated with the required retargeting. Thus, although theoretically possible,the above-described problems make dynamic and portable testing unfeasible.

SUMMARY

Various deficiencies in the prior art are addressed by scheduling of scan chain access.

In one embodiment, an apparatus includes a processing module, a reordering buffer module, and a vector transformation module. The processing module is configured to receive a plurality of test operations associated with a plurality of segments of a unit under test and to generate therefrom input test data configured to be applied to the unit under test via a Test Access Port (TAP). The reordering buffer module is configured to receive the input test data from the processing element and to buffer the input test data in a manner for reordering the input test data to compose an input

test vector tor a scan chain of the unit under test. The vector transformation module is configured to receive the input test vector from the reordering buffer module and to apply a vector transformation for the input test vector.

In one embodiment, a non-transitory computer-readable storage medium stores instruction which, when executed by a processor, cause the processor to perform a method including receiving a plurality of test operations associated with a plurality of segments of a unit under test, generating, from the test operations, input test data configured to be applied to the unit under test via a Test Access Port (TAP), buffering the input test data in a manner for reordering the input test data to compose an input test vector for a scan chain of the unit under test, and applying a vector transformation for the input test vector.

In one embodiment, a method including receiving a plurality of test operations associated with a plurality of segments of a unit under test, generating, from the test operations, input test data configured to be applied to the unit under test via a Test Access Port (TAP), buffering the input test data in a manner for reordering the input test data to compose an input test vector for a scan chain of the unit under test, and applying a vector transformation for the input test vector.

BRIEF DESCRIPTION OF THE DRAWINGS

The teachings presented herein can be readily understood by considering the following detailed description in conjunction with the accompanying drawings, in which:

FIG. 1 depicts a high-level block diagram of a system testing environment including a testing system and a system under test;

FIG. 2 depicts a high-level block diagram of one embodiment of the testing system of FIG. 1, including a test generation tool and a software compiler cooperating to generate test instructions for a system under test;

FIG. 3 depicts a high-level block diagram of one embodiment of the testing system of FIG. 1, including a test generation tool and a software compiler cooperating to generate test instructions for a system under test;

FIGs. 4A - 4E depict an implementation of the TISA using a SPARC V8 ISA, illustrating the details of instruction coding for the implementation of the TISA using a SPARC V8 ISA;

FIG. 5 depicts an implementation of the TISA using a SPARC V8 ISA, illustrating an exemplary TISA architecture for implementation of the TISA using a SPARC V8 ISA;

FIG. 6 depicts an embodiment of a TISA-based testing environment supporting interactive testing capabilities;

FIG. 7 depicts an exemplary implementation of the TISA-based testing environment of FIG. 6;

FIG. 8 depicts an exemplary program architecture for performing optimization of the transmitter-receiver channel of the system under test of FIG. 5A;

FIG. 9 depicts one embodiment of a method for adapting an Instruction Set Architecture (ISA) flow of a processor to form a Test Instruction Set Architecture (TISA) flow;

FIG. 10 depicts one embodiment of a method for generating instructions adapted for use in testing at least a portion of a system under test;

FIG. 11A depicts one embodiment of a method for generating instructions adapted for use in testing at least a portion of a system under test;

FIG. 11B depicts one embodiment of a method for generating instructions adapted for use in testing at least a portion of a system under test;

FIG. 12 depicts an exemplary embodiment of a TISA processor architecture;

FIG. 13 depicts an exemplary embodiment of a test processor architecture utilizing multiple processors to provide system testing capabilities;

FIG. 14 depicts an exemplary embodiment of a test co-processor architecture;

FIG. 15 depicts an exemplary embodiment of a test adjunct processor architecture;

FIG. 16 depicts an exemplary register set that can be used by a TISA processor;

FIG. 17 depicts a high-level block diagram of a system under test, illustrating an exemplary decomposition of an exemplary scan chain of the system under test;

FIG. 18 depicts a high-level block diagram of one embodiment of a method for testing a portion of a system under test via a scan chain of the system under test using Scan Segment Level abstraction of the scan chain;

FIG. 19 depicts one embodiment of a scheduler execution architecture configured to provide position-based scheduling for testing a system under test;

FIG. 20 depicts an exemplary TISA segment concatenation for an exemplary scan chain composition;

FIG. 21 depicts an exemplary scan chain composition and associated Circuit Model for the system under test of FIG. 19;

FIG. 22 depicts one embodiment of a process for performing position-based scheduling of access requests using the concept of critical section;

FIG. 23 depicts one embodiment of a scheduler execution architecture configured to provide position-based scheduling for testing a system under test;

FIG. 24 depicts an exemplary scan chain composition and associated Circuit Model where the scan chain composition includes a dynamic segment;

FIG. 25 depicts one embodiment of a method for supporting position-based scheduling for testing a system under test;

FIG. 26 depicts one embodiment of a scheduler execution architecture configured to provide position-based scheduling for testing a system under test;

FIG. 27 depicts one embodiment of a method for supporting position-based scheduling for testing a system under test;

FIG. 28 depicts one embodiment of a deferred scheduler execution architecture configured to provide position-based scheduling for testing a system under test;

FIG. 29 depicts one embodiment of a scheduler execution architecture configured to provide position-based scheduling for testing a system under test; and

FIG. 30 depicts a high-level block diagram of a computer suitable for use in performing functions described herein.

To facilitate understanding, identical reference numerals have been used, where possible, to designate identical elements that are common to the figures.

DETAILED DESCRIPTION

Various system testing capabilities are provided for use in performing testing of a system under test (SUT).

In one embodiment, a test instruction set architecture (TISA) is provided. The TISA is provided for use in performing system testing. The TISA combines computer science capabilities with system testing capabilities to provide improved system testing capabilities, including interactive testing capabilities, remote testing capabilities, and various other capabilities described herein. The TISA is formed by adapting a software-based instruction set architecture (ISA) using system testing capabilities. The software-based ISA may utilize any suitable software programming language (e.g., C++, Java, and the like, as well as various combinations thereof) and may be implemented using any suitable processor. The system testing

capabilities may utilize any suitable TAP, such as IEEE 1149.1 (also known as JTAG) TAPs or any other suitable TAPs. In general, the TISA is formed by combining the atomic operations of a software process with atomic testing operations of a test procedure. In the TISA, the algorithmic portions of the test procedure are handled by the software flow, such that the algorithmic portions of the test procedure are translated into the atomic testing operations. The TISA is formed by combining the atomic operations of the software process with the atomic testing operations of the test procedure, such that the atomic testing operations are treated in the same manner as the atomic operations of the software process that is handling the algorithmic portions of the test procedure. This enables finer-grain control of embedded test execution, remote test execution, and various other improved system testing capabilities as depicted and described herein.

FIG. 1 depicts a high-level block diagram of a system testing environment including a testing system and a system under test.

As depicted in FIG. 1, system testing environment 100 includes a testing system (TS) 110 and a system under test (SUT) 120.

The TS 110 may be any system suitable for testing SUT 120. The TS 110 is configured for testing SUT 120. The TS 110 may perform any testing of SUT 120, e.g., testing one or more individual components of SUT 120, one or more combinations of components of SUT 120, one or more interconnections between components of SUT 120, one or more system level functions of SUT 120, and the like, as well as various combinations thereof. The TS 110 may perform any of the functions typically associated with testing a system under test, such as executing test procedures, providing input data to the system under test, receiving output data from the system under test, processing output data received from the system under test for determining system testing results, and like functions, as well as various combinations thereof. The design and use of TS 110 for testing a system under test is described in additional detail hereinbelow.

The SUT 120 may be any system which may be tested using TS 110. The SUT 120 may include any component(s), at least a portion of which may be tested, individually and/or in combination, by TS 110. The TS 120 may include one or more scan chains, having one or more sets of associated input and output access pins, providing access to the component(s) to be tested by SUT 120. The manner in which a scan chain(s) may be utilized in SUT 120 for testing SUT 120 will be appreciated by one skilled in the art. For example, SUT 120 may include one or more boards, testing of which may be performed using one or more scan chains having associated input and output access pins which may be used for applying input testing signals to SUT 120 and collecting output testing signals from SUT 120.

As depicted in FIG. 1, TS 110 accesses SUT 120 via a test access interface (TAI) 115. The test access interface may be implemented using any suitable test access interface, which may depend on one or more of the TS 110, the SUT 120, the type of testing to be performed, and the like, as well as various combinations thereof.

For example, TAI 115 may include a Joint Test Action Group (JTAG) Test Access Port (TAP) as standardized in IEEE 1149.1 standard, which is incorporated by reference herein in its entirety. The IEEE 1149.1 standard defines a TAP that supports the following set of signals: Test Data In (TDI), Test Data Out (TDO), Test Mode Select (TMS), Test Clock (TCK), and, optionally, Test Reset Signal (TRST). The TDI and TDO pins of SUT 120 are interconnected in a boundary scan chain by which TS 110 may access SUT 120 for testing at least a portion of SUT 120.

The TAI 115 may include any other suitable test access interface.

It will be appreciated by one skilled in the art that TS 110, TAI 115, and SUT 120 may be implemented in any manner suitable for providing features of the embodiments covered herein.

As described herein, the TISA is able to leverage computer science capabilities in combination with system testing capabilities to provide a

significant improvement in system testing. A general description of system testing capabilities and computer science capabilities follows, followed by a description of the manner in which computer science capabilities and system testing capabilities may be utilized together to provide the TISA.

The TISA improves upon system testing capabilities by leveraging computer science capabilities. The system testing capabilities may include the capabilities generally supported in all stages of the "automated test" flow (which generally includes all of the steps and resources that may be needed to get from a definition of the test algorithm(s) to actual testing operations).

In order to help test automation, test resources often are embedded inside the boards and devices, and can be accessed using a standardised interface, usually called the Test Access Port (TAP). This has the effect of limiting the pin count and rationalising resource access and management. A number of languages are available for describing resources inside a system under test, and, thus, which may be used as inputs to Test Generation Tools (TGTs). TGTs can apply algorithms to generate testing sequences which may be used by a Test Control Unit (TCU) to command the TAP and execute the associated testing operations. The features and performances of the testing operations depend on these three elements: the access standard, the data format, and the TCU implementation.

The TISA is able to leverage computer science capabilities to provide improved system testing capabilities. This may include use of computer science capabilities that are available in all stages of the "software development flow" (which generally includes any or all of the steps and resources that may be needed to get from a software algorithm coded in a software language(s) of choice to the final debugging and execution on a target processor, such as compilation, an Instruction Set Architecture (ISA), interactive debugging, and the like, as well as various combinations thereof).

The use of compilation in computer science reduces an algorithm defined in a programmer-friendly high level abstraction to a series of machine-

executable instructions. This process can vary greatly, depending on the input programming language and project complexity; however, most, if not all, of the approaches share the same basic assumption: any algorithm can be decomposed into basic instructions, regardless of its complexity. This applies to classic languages, as well as to more modern high-level and object oriented languages such as, for example, C++, Java, Python, and the like.

The Instruction Set Architecture (ISA) is the core of any processor, and the reason for which compilation is so effective. In general, each processor offers a set of instructions which define the manner in which the processor can be operated. The instructions form at least part of the ISA of the processor. It will be appreciated that the ISA may be considered to include various constructs associated with the instructions, such as registers, addressing modes, opcodes, memory structures, and the like, as well as various combinations thereof. The ISA enables the processor to execute simple instructions, such as reading/writing values from/to memory, perform logical or arithmetical operations on registers, handle interruption, and the like. This basic behaviour has remained essentially unchanged over time, and modern processors achieve exceptional performances because they can efficiently exploit great numbers of resources, and, thus, are able to complete a much larger number of such basic instructions in approximately the same amount of time. Furthermore, even higher performances may be reached from the use of co-processors (e.g., floating-point co-processors, graphical co-processors, and the like), which can help the main processor by hard-coding complex operations.

The use of debugging in computer science allows monitoring and verification of the software development and execution process. In general, software development is a long and difficult process, which is strictly monitored and verified to assure that the final product is free of defaults, or "bugs" are they are usually called. In order to help test software programs, the software development flow provides many powerful debug features. For

example, common software development flow debug features include step-by-step execution; observability/controllability of all registers and memory locations, use of breakpoints and watchpoints, and the like. These debug features, as well as various other debug features, are more often enabled by algorithms and structures embedded into the final code by the software compiler, but may also be assisted by hardware resources available inside of the processor,. From this information the debugger can reconstruct the original code and correlate all the ISA-level operations to the programming abstraction layer.

The use of automated test execution capabilities and computer science software capabilities together to enable improved system testing capabilities may be better understood by way of reference to FIG. 2 and FIG. 3.

FIG. 2 depicts a high-level block diagram of one embodiment of the testing system of FIG. 1, including a test generation tool and a software compiler cooperating to generate test instructions for a system under test.

As depicted in FIG. 2, the TS 110 includes a test generation tool (TGT) 210 and a software compiler (SC) 220.

The TGT 210 includes a TGT composer 212 and TGT algorithms 214.

The TGT composer 212 accepts system description files 211 as input. The system description files 211 include any suitable description files which may be used by a TGT to produce testing instructions/vectors for testing a system under test. For example, system description files 211 may include circuit description files, board/fixture netlist files, other description files, and the like, as well as various combinations thereof. The system description files 211 may be available on TGT 210 and/or may be obtained from one or more remote components and/or systems.

The system description files 211 may include one or more circuit description files, The circuit description files may be specified using any suitable description language(s), such as the Boundary Scan Description Language (BSDL, which was developed as part of the IEEE 1149.1 standard

tor board-level JTAG), the Hierarchical Scan Description Language (HSDL, which was developed as an extension of BSDL), New Scan Description Language (NSDL), and the like, as well as various combinations thereof.

The system description files 211 may include one or more board/fixture netlist files, The board/fixture netlist files may include files related to the physical description of the device(s), describing the netlist, connections, and like information. The board/fixture netlist files may be specified in any suitable format, such as PCB, Gerber, and/or any other format suitable for board/fixture netlist files.

The system description files 211 may include one or more other description files. The other description files may include any other suitable description files which may be used as input for producing a circuit model. For example, other description files may include any suitable application-specific and/or tool-specific description language files, such as Asset's Macro Language, Goepel's CASLAN Language, and/or any other suitable description language files.

The TGT composer 212 processes the system description files 211 to produce a circuit model 213. The processing of system description files 211 by TGT composer 212 to produce circuit model 213 may be performed in any suitable manner. The circuit model 213 specifies a model of the system under test or portion of the system under test for which TGT 210 is being run. The TGT composer 212 provides circuit model 213 to TGT algorithms 214.

The TGT algorithms 214 accept circuit model 213. The TGT algorithms 214 process the circuit model 213 to produce TGT atomic test operations 216. The processing of circuit model 213 by TGT algorithms 214 to produce the TGT atomic test operations 216 may be performed in any suitable manner.

The SC 220 includes SC front-end algorithms 222 and SC back-end algorithms 224.

The SC front-end algorithms 222 accept computer science source files 221 as input. The computer science source files 221 include any suitable

computer science source files which may be compiled by a compiler. For example, computer science source files 221 may include computer science source files for any suitable computer programming language(s), such as C++, Java, Python, and the like, as well as various combinations thereof. For example, computer science source files 221 may include one or more of one or more C files, one or more C++ files, and/or any other suitable computer science source files.

The SC front-end algorithms 222 process the computer science source files 221 to produce a program model 223. The program model 223 specifies an intermediate representation of the computer science source files 221. The SC front-end algorithms 222 provide the program model 223 to the SC back-end algorithms 224.

The SC back-end algorithms 224 accept program model 223 as input. The SC back-end algorithms 224 process the program model 223 to produce one or more ISA Binary Files 225 including ISA atomic operations 226. The processing of program model 223 by the SC back-end algorithms 224 to form the ISA Binary Files 225 including the ISA atomic operations 226 may be performed in any suitable manner. The ISA atomic operations 226 are assembly-level instructions supported by the processor for which the TISA is implemented.

As depicted in FIG. 2, in addition to the respective processing flows of TGT 210 and SC 220, additional interaction between TGT 210 and SC 220 may be utilized for controlling generation of the TISA atomic operations 235. In one embodiment, SC back-end algorithms 224 may initiate one or more vector computation requests 230 to TGT algorithms 214. The SC back-end algorithms 224 may initiate a vector computation request 230 when the SC back-end algorithms need to access the TAP. The TGT algorithms 214, upon receiving a vector computation request 230 from SC back-end algorithms 224, generate one or more TGT atomic test operations 216 for the TAP based on the received vector computation request 230. The one or more TGT atomic

test operations 216 may then be applied to the TAP in a manner controlled by SC back-end algorithms 224, because the TGT atomic test operations 216 are combined with the ISA atomic operations 226 to enable algorithmic control over TGT atomic test operations 216 using ISA atomic operations 226. In this manner, the SC 220 provides algorithmic control of access to the TAP.

As depicted in FIG. 2, in addition to TGT 210 and SC 220, TS 110 further includes a TISA composer 240. The TISA composer 240 accepts the TGT atomic test operations 216 and the ISA atomic operations 226. The TISA composer 240 converts the TGT atomic test operations 216 into TISA instructions and inserts the TISA instructions into the ISA Binary File(s) 225 (i.e., combining the TISA instructions with the ISA atomic operations 226 to form thereby TISA Binary files 245 including TISA atomic operations 246. The TISA composer 240 may be part of TGT 210, part of SC 220, split across TGT 210 and SC 220, implemented separate from TGT 210 and SC 220, and the like.

It will be appreciated that the various inputs and outputs depicted and described with respect to FIG. 2 may be stored, displayed, executed, propagated, and/or handled in any other suitable manner, as well as various combinations thereof.

FIG. 3 depicts a high-level block diagram of one embodiment of the testing system of FIG. 1, including a test generation tool and a software compiler cooperating to generate test instructions for a system under test.

As depicted in FIG. 3, TS 110 of FIG. 3 operates in a manner similar to TS 110 of FIG. 2, in that TISA Binary files including TISA atomic operations are generated using interaction between the test generation tool and the software compiler; however, interaction between the test generation tool and the software compiler in TS 110 of FIG. 3 is different than interaction between the test generation tool and the software compiler in TS 110 of FIG. 2.

As depicted in FIG. 3, the TS 110 includes a test generation tool (TGT) 310 and a software compiler (SC) 320.

The TGT 310 includes a TGT composer 312 and TGT algorithms 314.

The TGT composer 312 accepts system description files 311 as input. The system description files 311 include any suitable description files which may be used by a TGT to produce testing instructions/vectors for testing a system under test. For example, system description files 311 may include circuit description files, board/fixture netlist files, other description files, and the like, as well as various combinations thereof. The system description files 311 of FIG. 3 may include system description files similar to system description filed 211 depicted and described with respect to FIG. 2 (e.g., one or more circuit description files, one or more board/fixture netlist files, one or more other description filed, and the like, as well as various combinations thereof). The system description files 311 may be available on TGT 310 and/or obtained from one or more remote components and/or systems.

The TGT composer 312 accepts one or more test operation description files 331_1 - 331_N (collectively, test operation description files 331) as input. The test operation description files 331 are generated by SC 320. The generation of test operation description files 331 by SC 320 is described in detail hereinbelow.

The TGT composer 312 processes the system description files 311 and the test operation description files 331 to produce a circuit model 313. The processing of system description files 311 by TGT composer 312 to produce circuit model 313 may be performed in any suitable manner. The circuit model 313 specifies a model of the system under test or portion of the system under test for which TGT 310 is being run. The processing of system description files 311 in conjunction with test operation description files 331 enables the TGT composer 312 to produce circuit model 313 in a manner for enabling TGT 310 to produce appropriate TAP atomic operations. The TGT composer 312 provides circuit model 313 to TGT algorithms 314.

The TGT algorithms 314 accept circuit model 313. The TGT algorithms 314 process the circuit model 313 to produce TGT atomic test operations 316.

The processing of circuit model 313 by TGT algorithms 314 to produce the TGT atomic test operations 316 may be performed in any suitable manner.

As depicted in FIG. 3, in addition to TGT 310 and SC 320, TS 110 includes a TISA translator 340. The TISA translator 340 receives the TGT atomic test operations 316. The TISA translator 340 translates TGT atomic test operations 316 to form TISA atomic test operations 346. The TISA translator 340 provides TISA atomic test operations 346 to SC 320 for inclusion in the software compilation process. The use of TISA atomic test operations 346 by SC 320 is described in detail hereinbelow. The TISA translator 340 may be part of TGT 310, part of SC 320, split across TGT 310 and SC 320, implemented separate from TGT 310 and SC 320, and the like.

The SC 320 includes a SC pre-compiler 330, SC front-end algorithms 322, and SC back-end algorithms 324.

The SC pre-compiler 330 accepts computer science source files 321.

The computer science source files 321 include any suitable computer programming source files which may be compiled by a compiler. For example, computer science source files 321 may include computer programming source files for any suitable computer programming language(s), such as C++, Java, Python, and the like, as well as various combinations thereof. IFor example, computer science source files 321 may include one or more of one or more C files, one or more C++ files, and/or any other suitable computer science source files.

The SC pre-compiler 330 processes the computer science source files 321.

The SC pre-compiler 330 processes the computer science source files 321, producing therefrom pre-processed computer science source files 321_P. The computer science source files 321 may be pre-processed by SC pre-compiler 330 to form pre-processed computer science source files 321_P in any suitable manner. The SC pre-compiler 330 provides the pre-processed computer science source files 321_P to front-end algorithms 322.

The SC pre-compiler 330 detects test operations during processing of the computer science source files 321, and generates the test operation description files 331. The test operation description files 331 may be specified using any suitable test description language (e.g., using one or more standard test description languages, using a test description language specific to the TGT 310, and the like, as well as various combinations thereof). The SC pre-compiler 330 provides the test operation description files 331 to TGT 310 (illustratively, to the TGT composer 312 of TGT 310, which processes the test operation description files 331 in conjunction with the system description files 311 to produce circuit model 313.

The SC front-end algorithms 322 accept pre-processed computer science source files 321_P. The SC front-end algorithms 322 also accept the TISA atomic test operations 346, which are produced by TISA translator 340 using TGT atomic test operations 316 produced by TGT 310 from the test operation description files 331. The SC front-end algorithms 222 compile the pre-processed computer science source files 321_P and TISA atomic test operations 346 to produce a program model 323. The program model 323 specifies an intermediate representation of the pre-processed computer science source files 321_P, which includes TISA atomic test operations 346 such that TISA atomic test operations 346 may be integrated within the ISA atomic operations to form TISA atomic operations. The SC front-end algorithms 322 provide the program model 323 to the SC back-end algorithms 324.

The SC back-end algorithms 324 accept program model 323. The SC back-end algorithms 324 process program model 223 to produce one or more TISA Binary Files 355 including TISA atomic operations 356. The processing of program model 323 by the SC back-end algorithms 324 to form the TISA Binary Files 355 including the TISA atomic operations 356 may be performed in any suitable manner.

The TISA atomic operations 356 include ISA atomic operations (i.e., assembly-level instructions supported by the processor for which the TISA is implemented) and TISA atomic test operations 346.

The TISA atomic operations 356 provide algorithmic control (using ISA atomic operations) over TGT atomic test operations 316 (i.e., in the form of the TISA atomic test operations 346), thereby enabling improved system testing of the system under test to which the TISA atomic operations 356 are to be applied. Thus, the TGT atomic test operations 316 (i.e., in the form of the TISA atomic test operations 346) may be applied to the TAP in a manner controlled by SC back-end algorithms 324, because the TGT atomic test operations 316 are combined with the ISA atomic operations to enable algorithmic control over TGT atomic test operations 316 using the ISA atomic operations. In this manner, the SC 220 provides algorithmic control of access to the TAP.

It will be appreciated that the various inputs and outputs depicted and described with respect to FIG. 3 may be stored, displayed, executed, propagated, and/or handled in any other suitable manner, as well as various combinations thereof.

With respect to FIG. 2 and FIG. 3, although primarily depicted and described with respect to specific numbers of input files, intermediate files, models, output files, and the like, it will be appreciated that the embodiments of FIG. 2 and FIG. 3, as well as various associated teachings provided herein, may be implemented using any suitable numbers of input files, intermediate files, models, output files, and the like.

FIG. 2 and FIG. 3 illustrate the manner in which computer science capabilities may be leveraged to improve system testing capabilities (e.g., providing finer-grain control of system testing, enabling interactive system testing, enabling interactive debugging during system testing, and providing various other advantages depicted and described herein). The system testing schemes of FIG. 2 and FIG. 3 provide improvements over existing

approaches, such as STAPL, where the goal is to add programming features to vector formats and, therefore, debugging, remote access, and interactivity features are added from scratch. By contrast, the TISA leverages the wealth of information from computer programming and embedded applications to control test access for system testing.

Referring to FIGs. 2 and 3, it will be appreciated that the capabilities and features of the TISA are defined by its abstraction level, i.e., the finer the definition of the TISA atomic operations, the better performance the TISA will provide.

In one embodiment, in which TISA is implemented in a JTAG architecture, three abstraction levels may be supported for scan operations.

The first abstraction level is the Vector Level. The Vector Level is the coarsest grain of the three abstraction levels, where the atomic operations are inputs and outputs of scan vectors. The Vector Level is best represented in a vector format, such as Serial Vector format (SVF) or any other suitable vector format, and gives the highest-level control.

The second abstraction level is the TAP Level. In the TAP Level, the atomic operations are enhanced to allow full control over the TAP state machine. This enables more refined control over scan operations, support of non-standard sequences (e.g., like the ones required, for instance, in the Addressable Shadow Protocol or other similar protocols).

The third abstraction level is the Scan Segments Level. The Scan Segments Level is the finest grain of the three abstraction levels. The Vector Level and TAP Level abstraction levels use the scan vector as the atomic data format, which is sufficient for traditional continuity tests where the entire scan chain is involved, but is cumbersome for instrument-based testing where there is a need for fine-grain control over the tens or hundreds of instruments that compose the scan chain. The Scan Segments Level allows the definition of "scan segments" inside the overall scan path, which can be handled separately, thereby providing a flexible and powerful set of primitives that can

be used to define scan operations directly in the problem space and resolve the scan operations at implementation time. This approach is advantageous in embedded applications, where the available computational resources may be quite limited. The use of Scan Segments Level is depicted and described in additional detail hereinbelow.

As depicted in FIG. 2 and FIG. 3, regardless of the abstraction level of the scan operations, the resulting TAP atomic operations (illustratively, TGT atomic test operations 216 and TGT atomic test operations 316) computed by the TGT are converted into corresponding TISA atomic test operations and inserted into the binary executable (i.e., into the ISA atomic operations generated by the SC).

Referring to FIG. 2, TGT atomic test operations 216 and ISA atomic operations 226 can be processed to form the TISA atomic operations 246 in the TISA binary executables (illustratively, TISA binary files 245). The TISA atomic operations 246 include TISA atomic test operations and ISA atomic operations.

Referring to FIG. 3, TISA atomic test operations (generated by TISA translator 340 from TGT atomic test operations 316 produced by TGT 310) can be input into the SC front end 324 as pre-compiled assembly instructions, without any need to modify the SC front end 324 of SC 310. It will be appreciated that almost all programming languages allow for such operations. In C, for example, this operation is obtained using the "asm" command. In one embodiment, minor modifications to SC back-end algorithms 324 may be required (e.g., to handle binary conversion of the TISA assembler instructions). An example of such a process is depicted and described herein with respect to FIG. 11.

Although primarily depicted and described with respect to levels of granularity of TISA atomic operations in a JTAG architecture, it will be appreciated by one skilled in the art that the same levels of granularity of TISA atomic operations may be utilized in other architectures, that different levels of

granularity of TISA atomic operations may be utilized in a JTAG architecture and/or other architectures, and the like, as well as various combinations thereof.

As described hereinabove, the TISA may be implemented using any suitable instruction set architecture (ISA). For example, the TISA may be implemented using the SPARC V8 ISA, an INTEL ISA, and the like.

For purposes of clarity in describing implementation of the TISA, an exemplary implementation of the TISA using a SPARC V8 ISA is depicted and described herein with respect to FIGs. 4A-4E. In this exemplary implementation, the TISA is implemented as a Vector Level TISA, which allows direct coding of the instructions that compose the SVF format; however, as described hereinabove, it will be appreciated that implementation of the TISA using the SPARC V8 ISA also may be performed where the TISA is implemented as a TAP Level TISA or a Scan Segment Level TISA.

The SPARC V8 ISA is implemented in many products, such as the open-source soft processor family Leon 2 and Leon 3.

A review of "The SPARC Architecture Manual Version 8," published by SPARC International, Inc, 1992 (hereinafter "SPARC Architecture Manual"), reveals that there are many code words not exploited by the SPARC V8 ISA. This is evident at least from a review of the "opcodes and condition codes" of Appendix F.

FIG. 4A depicts the unexploited code words of the SPARC V8 ISA. The unexploited code words depicted in FIG. 4A may be used to code the "test" instructions for the TISA. More specifically, when both "op" and "op2" are set to 0, the instruction is marked as unimplemented in "The SPARC Architecture Manual Version 8," such that the instruction may be used for the TISA.

FIG. 4B depicts a coding format able to represent all thirteen of the SVF instructions. As depicted in FIG. 4B, bits 30-25 include the instruction coding itself, bits 21-18 may be used to code a TAP state if one is to be used

with the instruction, and bits 17-14 can be used by each instruction to specify optional information where needed.

FIG. 4C depicts an exemplary bit coding of the TAP states for an IEEE 1149.1 TAP. The bit coding of the TAP states is represented using a first column that identifies the IEEE 1149.1 TAP State Name, a second column that identifies the SVF TAP State Name associated with the IEEE 1149.1 TAP State Name, and a third column that identifies the bit coding for bits 21-18 of FIG. 4B. It will be appreciated that the bit codings may be assigned to the TAP states in various other ways.

The SVF instructions allow for multiple parameters, which need to be coded inside the final code. In order to represent the parameters, and in the interest of the usual architectural best practice of keeping instruction and data separated, register-based parameter passing is defined for this exemplary implementation of a Vector Level TISA. Thus, the Vector Level TISA presents six dedicated 32-bit registers: GENERIC1 , GENERIC2, TDI, TDO, MASK and SMASK. The six dedicated 32-bit registers are depicted in FIG. 4D. The usage of the six dedicated 32-bit registers is described in detail hereinbelow, but, as a general rule, these registers are used either to store a parameter or to point to the memory location in which a parameter is stored. Thus, at compilation time, normal ISA instructions can be used to load these registers before the TISA instruction is invoked. More specifically, in this SPARC V8 ISA implementation of the TISA, coprocessor registers may be used directly as parameters for the usual load/store instructions.

The SVF instructions which may be utilized in this SPARC V8 ISA implementation of the TISA include ENDDR, ENDIR, STATE, FREQUENCY, PIO, PIOMAP, HDR, HIR, TDR, TIR, SDR, SIR, and RUNTEST. These SVF instructions may be better understood by way of reference to the "Serial Vector Format Specification," by ASSET InterTech, Inc., 1997 (hereinafter referred to as the SVF Manual), which is herein incorporated by reference in

its entirety. The use of these SVF instructions in this SPARC V8 ISA implementation of the TISA is described in more detail hereinbelow.

ENDDR. ENDIR, STATE

The ENDDR and ENDIR instructions indicate the TAP state at which the TAP interface ends its operation. The STATE instruction forces the TAP interface to a specified state. In this exemplary implementation of the TISA, the SVF codings for the ENDDR, ENDIR, and STATE instructions are "000000", "000001 ", and "00001 0", respectively, as depicted in FIG. 4E. The SVF coding of these SVF instructions may be performed using the "TAP STATE" file (i.e., the exemplary bit coding of the TAP states as depicted in FIG. 4C) as needed. It will be appreciated, at least from a review of the SVF Manual, that the STATE instruction can optionally take the explicit sequence of states as parameters. In this exemplary implementation of the TISA, taking the explicit sequence of states as parameters would be coded by a series of instructions, one for each state in the sequence.

FREQUENCY

The FREQUENCY instruction is used to specify the working frequency of the TAP interface. The FREQUENCY instruction is expressed as a 32-bit integer of Hz cycles. In this exemplary implementation of the TISA, the SVF coding for the FREQUENCY instruction is "00001 1", as depicted in FIG. 4E. The value for the FREQUENCY instruction is stored in the GENERIC1 register.

PIO. PIOMAP

The PIO instruction can be used to handle parallel vectors, in a format previously set by a call to PIOMAP. In this exemplary implementation of the RISA, PIOMAP is seen as a pre-processor directive that generates the appropriate commands to set up the TAP interface. Thus, the PIO instruction

merely needs to express the parallel vector, which can be expressed by indicating (in the GENERIC1 register) the address in which the parallel vector is stored. The number of words "n" that compose the vector is specified in bits 13-0 of the instruction, and, thus, the vector has an upper size limit of 2^{13} = 8K words = 32 Kbytes. If the vector size is not an exact multiple of a word, padding and re-alignment may be provided in memory, as needed. In this exemplary implementation of the TISA, the SVF coding for the PIO instruction is "0001 00".

HDR, HIR. TDR, TIR

The roles of the HDR, HIR, TDR, and TIR instructions are different. Here, these SVF instructions are considered together because (1) these SVF instructions are functionally similar (i.e., they all command shift operations, even if they are of a different nature), and (2) these SVF instructions accept the same parameters:

(1) length: a 32-bit number expressing the number of bits to shift;

(2) TDI (optional): the input shift vector;

(3) TDO(optional): the expected output shift vector;

(4) MASK (optional): a mask to be used when comparing actual values with TDO. A 'l ' indicated a care, a 'O' a don't care; and

(5) SMASK (optional): a mask to mark which bits are to be considered in TDI. 'l ' indicates a care, 'O' a don't care.

In this exemplary implementation of the TISA, the SVF codings for the HDR, HIR, TDR, and TIR instructions are "0001 10", "0001 11", "001 0 10", and "001 0 11", respectively, as depicted in FIG. 4E.

In this exemplary implementation of the TISA, the following additional codings may be used:

(1) length is stored in the GENERIC1 register;

(2) 0 1 is '1' when TDI is present, 'O' otherwise. If set, the TDI register contains the address at which the input vector is stored;

(3) 0 2 is 'l ' when TDO is present, 'O' otherwise. If set, the TDO register contains the address at which the expected output is stored;

(4) 0 3 is T when MASK is present, 'O' otherwise. If set, the MASK register contains the address at which the output mask is stored; and

(5) 0 4 is 'l ' when SMASK is present, 'O' otherwise. If set, the SMASK register contains the address at which the output mask is stored.

SDR. SIR

The SDR and SIR instructions have the same syntax as the HDR, HIR, TDR, and TIR instructions, but have a functional difference: SDR and SIR trigger the actual scan operation on the TAP. In interactive testing the actual output vector read from the system is fundamental for the algorithm, so the TISA offers the possibility of storing the actual output vector in memory. When the "TAP STATE" field (bits 21-18, as depicted in FIG. 4B) is different than zero, the GENERIC2 register indicates the storage location of the actual output vector. Thus, SDR and SIR can support a maximum of seven parameters. If TDO is specified and the actual output vector is different from the expected output vector, an overflow flag is set in the Processor State Register (PSR), as described in Section 4.2 of the SPARC Architecture Manual.

RUNTEST

The RUNTEST instruction forces the TAP interface to run a test at a specified state for a specified amount of time, and is used mainly to control RUNBIST operations (e.g., as defined in IEEE 1149.1). The RUNTEST instruction accepts one or more of the following parameters (all of which are optional):

(1) run_state: the state the interface must maintain during test execution;

(2) run_count: the number of clock cycles the test must take;

(3) run_clk: which clock run_count refers to(TCK: TAP clock, SCK: system clock);

(4) min_time: minimum run time in seconds, expressed as a real number;

(5) max_time: maximum run time in seconds, expressed as a real number; and

(6) endstate: the state the interface must reach at the end of the command.

In this exemplary implementation of the TISA, the SVF coding for the RUNTEST instruction may be "0001 0 1" or "1001 0 1".

In this exemplary implementation of the TISA, the following additional codings may be used:

(1) TAP_STATE: it contains run_state of which it is defined;

(2) 0 1: '1' if TAP_STATE is defined, 'O' otherwise;

(3) 02: T if min_count is specified, 'O' otherwise. If set, the GENERIC1 register contains the 32-bit unsigned representation of min_count;

(4) 03: T if maxjime is set, 'O' otherwise. If set, the GENERIC2 register contains the 32-bit unsigned representation of max_count;

(5) 04: '1' if endstate is set, 'O' otherwise. If set, Bits 13-1 0 contain the end state.

(6) Bits 9-0: if run_count is specified, expressed as an unsigned integer (max run_count=2^{10}=1024). If this field is not "0", then Bit 30 indicates run_clock ('I '=TCK, O'=SCK).

Although primarily depicted and described herein with respect to use of specific SVF instructions in this SPARC V8 ISA implementation of the TISA (i.e., namely, ENDDR, ENDIR, STATE, FREQUENCY, PIO, PIOMAP, HDR, HIR, TDR, TIR, SDR, SIR, and RUNTEST), it will be appreciated that fewer or more SVF instructions may be used.

Although primarily depicted and described herein with respect to an implementation of the TISA using the SPARC V8 ISA, it will be appreciated

that various other ISAs may be utilized to implement a TISA in accordance with the TISA teachings depicted and described herein.

In interactive testing approaches, the data handoff point is quite important. As described hereinabove, a test program is composed of two main portions: the algorithmic portion (as represented by the software compiler) and the test access portion (as represented by the test generation tool). During a test operation using a testing program, there will be moments when the test program is accessing the system under test, and moments when the test program is examining the testing results and deciding the next step(s) required. The hand-off between these two operations is important for obtaining efficient interactive testing.

In existing script-based approaches, such as SVF and STAPL, a script takes care of all TAP operations at the Vector Level. At this level, the interface (or "player") is able to communicate with the TAP protocol, and send/receive vectors to/from the system under test. Furthermore, STAPL also allows some basic flow control (if-then-else) and algorithmic operations on the bit vectors. If there is need for more sophisticated processing (e.g., identifying a register inside a received vector, or computing the vector to access a specific device), the player hands control over to the algorithmic portion. In STAPL, this is done through the "export" command. Disadvantageously, however, neither SVF nor STAPL has a standardized format for this (e.g., in the case of STAPL, the handoff process is usually proprietary to a given vendor).

In existing embedded approaches, like Master Test Controller (MTC) from Ericsson and the System BIST Processor, the same partitioning between the algorithmic portion and the test access portion is used. In such embedded approaches, the algorithmic portion and the test access portion are executed by different coprocessors that must be programmed separately. Furthermore, the memory spaces of the algorithmic portion and the test access portion are physically different, such that the resulting handoff mechanisms are similar to the handoff mechanisms of STAPL. The result is that the coprocessor for the

test access portion is forced to store a lot of scan operations before handoff to the algorithmic portion, which, given the increasing size of scan chains, may require a huge amount of resources.

In contrast with existing approaches to integrated testing (e.g., script-based approaches such as SVF and STAPL, and embedded approaches such as MTC and System BIST Processor), the TISA integrates the test access portion (i.e. the test operations) inside the algorithmic portion (i.e., the classical ISA), such that the test access portion and the algorithmic portion share the same physical memory space, thereby making handoff (and, thus, data passing) between the test access portion and the algorithmic portion automatic. In TISA, handoff between the test access portion and the algorithmic portion is made at the instruction level, such that the processor can freely mix scan and algorithm (i.e., freely mix test operations and algorithmic operations) as required according to the associated scheduling strategy.

In this exemplary implementation of the TISA, using the SPARC V8 ISA, all operations handling vectors use absolute addressing (as described hereinabove with respect to the SVF instructions). As a result, testing vectors may be used like normal variables inside the ISA program, thereby making the interface between the test access portion and the algorithmic portion automatic. As an example, based on the exemplary implementation of the TISA using the SPARC V8 ISA as described hereinabove, the following steps exemplify an archetypical testing sequence:

(1) An SDR instruction is used to obtain testing output data from the system under test. The resulting output data is places in a specific memory location (e.g., the "actual" parameter in the GENERIC2 register);

(2) A classical LOAD instruction can transfer this output data to be loaded into a register;

(3) Once the output data is loaded in the register, arithmetic operations and/or logical operations may be used to process the output data (note that

since the SPARC V8 ISA is a load/store architecture, all data must be loaded into a register before being handled);

(4) A classical STORE instruction is used to transfer the result of the algorithm into memory; and

(5) An SDR instruction can send new testing input data to the TAP (e.g., using the "TDI" parameter in the TDI register).

Note that the classical algorithmic operations (2) through (4) are standard for any ISA algorithm implementation, and are not modified in any way by the TISA.

Thus, from this simple example, it is clear that TISA can be supported using any given algorithm or computer program, with a natural and efficient hand-off between the algorithmic portion and the test access portion.

In this exemplary implementation of the TISA, using the SPARC V8 ISA, absolute addressing is used (for purposes of clarity in describing the TISA); however, one skilled in the art and informed by the teachings herein would be able to modify this exemplary implementation of the TISA to support all legal SPARC V8 addressing modes described in the SPARC Architecture Manual.

Although primarily depicted and described herein with respect to an exemplary implementation of the TISA in which SVF is used, SVF was used in the exemplary implementation because it is a well-known format proven to provide a complete, even if basic, handling of 1149.1 TAPs. It will be appreciated, by one skilled in the art and informed by the teachings herein, that the TISA may be implemented using any other suitable control formats, many of which may allow finer grain control of the TAP state machine and support more sophisticated testing operations.

Although primarily depicted and described herein with respect to an exemplary implementation of the TISA in which the abstraction level is the Vector Level, it will be appreciated, by one skilled in the art and informed by the teachings herein, that the exemplary TISA implementation depicted and

described herein may be modified such that the abstraction level of the TISA is the TAP Level or the Scan Segment Level.

For purposes of clarity in describing the TISA, an exemplary use of the TISA to perform testing on an exemplary system under test is depicted and described herein with respect to FIGs. 5 and 6. In this exemplary use of the TISA, the TISA is implemented as a Vector Level TISA using a SPARC V8 ISA and SVF (i.e., in continuation of the exemplary implementation depicted and described with respect to FIGs. 4A - 4E).

FIG. 5A and FIG. 5B depicts an exemplary use of the TISA to perform testing on a system under test.

FIG. 5A depicts a system test environment 500 including a JTAG TAP 510 and a system under test 520.

The JTAG TAP 510 provides test access to a system under test 520. The JTAG TAP 510 provides test access to the system under test 520, for sending input data to system under test 520 and receiving output data from system under test 520. The JTAG TAP 510 includes an instruction register (IR) 512, which is an 8-bit instruction register.

The JTAG TAP 510 is controlled by a testing system (e.g., such as testing system 110 depicted and described with respect to FIG. 3, which is omitted for purposes of clarity).

The system under test 520 includes a first board 521 (denoted as B1) and a second board 525 (denoted as B2). The first board 521 includes a transmitter 522 (denoted as T). The second board 525 includes a receiver 526 (denoted as R). The transmitter 522 sends data, on a connection 529, to receiver 526. In this example, the connection 529 is an 8-bit connection.

As depicted in FIG. 5A, each board is accessible from JTAG TAP 510 via its own scan chain. Namely, first board 521 is accessible via a first scan chain 523 and second board 525 is accessible via a second scan chain 527. The first scan chain 523 and second scan chain 527 are selectable by the IR 512 of JTAG TAP 510 (e.g., IR=0 selects first board B1, IR=1 selects second

board B2). The transmitter 522 and the receiver 526 are not alone on their boards; rather, they are part of wider scan chains (e.g., for purposes of this example, 24 bits and 16 bits, respectively).

In a test program, input data is sent to transmitter 522 via the first scan chain 523, and the resulting output data is collected from the receiver 526 by exploiting the second scan chain 527. In order to perform an exhaustive test, all possible values are sent through the connection 529, such that 2^8=256 vectors are sent through the connection 529. Using C, an exemplary program could be the following:

```
1  include <stdio.h>
2  include <jtag.h>
3
4  char sent_value, received value;
5
6  define MAX_COUNT 256;
7
8  void main(void)
9  {
10   for (sent_value=0;sent_value<MAX_COUNT;sent_value++)
11    {
12      apply_JTAG(sent_value,B1 .T);
13      read_JTAG (received_value,B2.R);
14       if (sent_value != received value) exit (0);
15    }
16  exit(1 );
17 }
```

In this program, line 2 includes the C module that is handling JTAG operations, where the functions "apply_JTAG" and "Read_JTAG", used in lines 12 and 13, respectively, are defined. The pre-compiler 330 of SC 320 recognizes these functions, and generates test operation description files 331 for TGT 310. The format of the test operation description files 331 may vary, depending on the actual implementation of first board 521 and second board 525. For example, if first board 521 and second board 525 both are IJTAG compliant, test operation description files 331 could be specified, for example,

using New Scan Description Language (NSDL) code. The TGT 310, using test operation description files 331, generates TGT atomic test operations 316, which are translated, by TISA translator 340, into TISA atomic test operations 346. The TISA atomic test operations 346 are provided to front-end 324 of SC 320. The TGT atomic test operations 316, the associated TISA atomic test operations 346, and the resulting TISA binary code are depicted in FIG. 5B.

FIG. 5B depicts a mapping from C commands to TISA coding for use by a testing system performing testing of the system test environment 500 of FIG. 5A.

As depicted in FIG. 5B, the mapping from C commands to TISA coding is represented using a table 540 having four columns: a "C command" column 541, an "SVF instructions" column 542, a "TISA assembler" column 543, and a "TISA coding" column 544. The table 540, from left to right, illustrates the manner in which a C command can be translated into an SVF instruction, which can be translated into TISA assembler, which can be coded into TISA binary coding.

The Apply_JTAG(value,B1 .T) command is translated into two SVF instructions: SIR 8 TDI(00) and SDR 24 TDI(value).

The SIR 8 TDI(00) SVF instruction is translated into TISA assembler as three operations:

SET 8, %cGENERIC1

SET 00, %cTDI

SIR TDI, which is translated into TISA coding as 1201 0000.

The SDR 24 TDI(value) SVF instruction is translated into TISA assembler as three operations:

SET 24, %cGENERIC1

SET value, %cTDI

SDR TDI, which is translated into TISA coding as 1001 0000.

The Read_JTAG(value,B2.R) command is translated into two SVF instructions: SIR 8 TDI(01) and SDR 16 ACTUAL(value).

The SIR 8 TDI(01) SVF instruction is translated into TISA assembler as three operations:

SET 8, %cGENERIC1

SET 0 1, %cTDI

SIR TDI, which is translated into TISA coding as 1201 0000.

The SDR 16 ACTUAL(value) SVF instruction is translated into TISA assembler as three operations:

SET 16, %cGENERIC1

SET "value", %cGENERIC2

SDR ACTUAL, which is translated into TISA coding as 10008000.

The TISA coding of the SET operations is not specified because the SPARC V8 Manual identifies them as "pseudo-instructions" which can have a different coding following the implementation of the processor.

Using the determined TISA codings, the pre-compiler 330 may now substitute the high-level JTAG accesses with their associated TISA assembler instructions. The result is the following code, specified using C, in which the calls to the JTAG TAP have been replaced by the associated TISA assembler coding:

```
1  include <stdio.h>
2  include <jtag.h>
3
4  char sent_value, received value;
5
6  define MAX_COUNT 256;
7
8  void main(void)
9  {
10   for (sent_value=0;sent_value<MAX_COUNT;sent_value++)
11    {
12      asm volatile ("SET 8, %cGENERIC1 ;
13                  SET 00, %cTDI;
```

```
14                SIR TDI;
15                SET 24, %cGENERIC1 ;
16                SET &sent_value, %cTDI;
17                SDR TDI;");
18      asm volatile ("SET 8, %cGENERIC1 ;
19                SET 01, %cTDI;
20                SIR TDI;
21                SET 16, %cGENERIC1 ;
22                SET &received_value, %cGENERIC2;
23                SDR ACTUAL");
24       if (sent_value != received value) exit (0);
25    }
26  exit(1);
27 }
```

This code can be input into the front-end algorithms 322, which will generate the program model 323. The program model 323 can be input into the back-end algorithms 324, which will generate the executable TISA binary file(s) 355 including the TISA atomic operations 356.

The "TISA coding" column 544 of table 540 depicts the binary coding of the TISA assembler instructions (e.g., using the various rules defined with respect to the exemplary implementation of the TISA using a SPARC V8 ISA, as depicted and described with respect to FIGs 4A - 4E).

As described herein, the TISA provides complete freedom regarding test granularity in performing testing of a system under test (i.e., from TAP Level through Scan Segment Level). As depicted in FIG. 2 and FIG. 3, and further explained using the exemplary TISA implementation of FIGs. 4A-4E and FIGs. 5A - 5B, test patterns may be computed using explicit queries by the Software Compiler to the Test Generation Tool, such that the only limit for the software algorithm is the resolution of the queries themselves.

As an example, at a coarse level, queries from the SC to the TGT may involve the entire scan chain of the system under test (e.g., such as in classical BSDL-based Boundary Scan testing).

As an example, at a fine level, queries from the SC to the TGT may involve registers or even bits. For example, dedicated Scan Segment primitives could significantly accelerate instrument access and TAP reconfiguration, boost code reuse, and provide various other advantages.

As an example, at a middle level somewhere between the coarse and fine levels, queries from the SC to the TGT may be done functionally (e.g., using standards such as IJTAG and other suitable standards, and using description languages such as NSDL and other suitable object-oriented description languages).

In this manner, the TISA does not force device/register access to be resolved at the model space (i.e., in the TGT), but, rather, allows developers to handle device/register access at the problem space (i.e., in the SC), thereby enabling developers to adapt the analysis grain to their needs and to the available resources.

Furthermore, in embodiments in which the TISA processor has sufficient resources, e.g., such as in the case of Automated Test Equipment (ATE), at least a portion of the circuit model may be implemented within the program model, thereby enabling the TISA machine to directly compute the vector patterns.

Furthermore, the TISA enables support for various other system test capabilities not previously possible without TISA, such as interactive testing including interactive debugging (locally and/or remotely), concurrency, portability, and the like, as well as various combinations thereof. These additional capabilities are now addressed in additional detail.

FIG. 6 depicts an embodiment of a TISA-based testing environment supporting interactive testing capabilities.

As depicted in FIG. 6, TISA-based testing environment 600 includes a host computer (HC) 601, a testing system (TS) 610, and a system under test (SUT) 620.

The HC 601 is configured to control TS 610 for controlling testing of SUT 620. The HC 601 includes a processor 602 coupled to a memory 604. The processor 602 and memory 604 may be any suitable processor and memory.

The memory 604 stores one or more debugger control programs 605. The debugger control program(s) enable HC 601 to trace and, where desired or necessary, alter, the execution of computer program(s) running on TS 610. For example, debugger control program(s) 605 may include one or more of the GNU Debugger (GDB), the dbx debugger, the Perl debugger, the Bash debugger, the Python debugger, and like suitable debugger programs, as well as various combinations thereof.

The memory 604 also may store one or more debugger display programs 606. The debugger display program(s) enable HC 601 to display information associated with the debugger control program(s) 605. The information associated with debugger control program(s) 605 may be displayed by debugger display program(s) 606 in any suitable manner (e.g., using one or more display devices). For example, debugger display program(s) 606 may include one or more of Insight (which is a graphical user interface to GDB), the Data Display Debugger (DDD, which provides a graphical user interface for various command-line debuggers, such as GDB and others), and like suitable debugger display programs, as well as various combinations thereof.

The TS 610 is controlled by HC 601 for purposes of testing SUT 620. The TS 610 is configured to function in a manner consistent with the TISA (e.g., such as depicted and described with respect to TS 110 of FIG. 1 - FIG. 3) and, further, is configured to support interactive testing (e.g., by enabling access by debuggers running on HC 601).

The TS 610 includes a TISA processor 612 coupled to a memory 614. The TISA processor 612 may be implemented using any suitable processor, such as SPARC V8 (as depicted and described hereinabove with respect to

FIGs. 4A-4E and FIG. 5), INTEL, and the like. The memory 604 may be any suitable memory.

The memory 614 stores one or more debugger program stubs 615. The debugger program stubs 615 understand the debugger protocol of the corresponding debugger control program(s) 605 running on HC 601, thereby enabling HC 601 to communicate with TS 610. For example, debugger stub(s) 615 may include one or more of GDB stub, a DBX stub, a Perl stub, a Bash stub, a Python stub, and like suitable debugger program stubs, as well as various combinations thereof.

The memory 614 stores TISA Binary Files 616. The TISA Binary Files 616 are generated by TS 610 in a manner as depicted and described herein with respect to FIG. 2 and FIG. 3. The TISA Binary Files 616 are executed by TISA processor 612 to perform testing on SUT 620.

The TS 610 also includes a Test Access Port (TAP) 618 coupled to TISA processor 612. The TAP 618 provides a test interface between TISA processor 612 and SUT 620 for enabling TISA processor 612 to perform testing of SUT 620 while being controlled by HC 601. The TAP 618 may be any suitable TAP (e.g., an 1149.1 TAP).

The TISA processor 612 interfaces with TAP 618 using an interface 617. The interface 617 may be any suitable interface between a TAP and a system under test (e.g., such as an interface that supports TCK, TMS, TDI, TDO, and, optionally, TRST, where TAP 618 is implemented as an 1149.1 TAP).

As depicted in FIG. 6, there is an interface 609 between HC 601 and TS 610. The interface 609 may support local communications and/or remote communications between HC 601 and TS 610. Thus, HC 601 may control interactive testing of SUT 620 via TS 610 locally and/or remotely.

For example, for local testing, interface 609 may be implemented as one or more of a Universal Asynchronous Receiver-Transmitter (UART)

interlace, serial interface, and the like, as well as various combinations thereof.

For example, for remote testing, interface 609 may be implemented using any suitable communications capabilities, such as Transmission Control Protocol (TCP) / Internet Protocol (IP) or any other suitable communications protocols. This enables remote testing in which the HC 601 and TS 610 may be separated by large geographical distances, and HC 601 will still be able to control TS 610 for purposes of performing testing of SUT 620.

In the TISA-based testing environment 600, the HC 601 is able to control, step-by-step, test execution on SUT 620, by controlling operation of TS 610 via a standard connection (e.g., UART, TCP/IP, and the like), thereby enabling interactive testing and debugging capabilities.

Although omitted for purposes of clarity, it will be appreciated that HC 601 and TS 610 may include various other components, such as additional processors, additional memories, internal communications buses, input/output modules, additional support circuits (e.g., power supplies), and the like, as well as various combinations thereof.

Although omitted for purposes of clarity, it will be appreciated that SUT 620 may be any system under test which may be tested using the TISA.

Although primarily depicted and described with respect to specific types of debugger control programs, debugger display programs, interfaces, and the like, it will be appreciated that TISA-based testing environment 600 may be implemented in a manner enabling fully-interactive testing capabilities using various other debugger control programs, debugger display programs, interfaces, and the like, as well as various combinations thereof.

FIG. 7 depicts an exemplary implementation of the TISA-based testing environment of FIG. 6.

As depicted in FIG. 7, exemplary TISA-based testing environment 700 of FIG. 7 is an implementation of the TISA-based testing environment 600 of

FIG. 6 in which the GNU Tool Suite is used to support interactive testing of the exemplary system testing environment 500 of FIG. 5A.

As depicted in FIG. 7, exemplary TISA-based testing environment 700 includes a host computer (HC) 701, a testing system (TS) 710, and a system under test (SUT) 720.

The HC 701 includes a processor 702 and a memory 704. The HC 701 of FIG. 7 is an implementation of HC 601 of FIG. 6, in which debugger control program(s) 605 is implemented using GDB (GDB 705) and debugger display program(s) 606 is implemented using DDD (DDD 706).

The TS 710 includes a TISA processor 712 and a memory 714. The TS 710 of FIG. 7 is an implementation of TS 610 of FIG. 6, in which the TISA processor 612 is implemented using a SPARC V8 ISA (denoted as SPARC V8 TISA processor 712), debugger program stub(s) 615 is implemented using a GDB stub (GDB stub 715), and the TISA Binary Files 616 are generated based on the SPARC V8 ISA associated with SPARC V8 TISA processor 712 (TISA Binary Files 716).

The TS 710 also includes a Test Access Port (TAP) 718 coupled to SPARC V8 TISA processor 712. The TS 710 of FIG. 7 is an implementation of TS 610 of FIG. 6, in which the TAP 618 is implemented using a 1149.1 TAP (1149.1 TAP 718).

The SPARC V8 TISA processor 712 interfaces with 1149.1 TAP 718 using an interface 717. The interface 717 is a standard 1149.1 interface that supports TCK, TMS, TDI, TDO, and, optionally, TRST.

The SUT 720 is the SUT 520 of FIG. 5A. The SUT 720 includes a transmitter and receiver on different boards, as in SUT 520 of FIG. 5A.

The 1149.1 TAP 718 provides a test interface between SPARC V8 TISA processor 712 and SUT 720 for enabling SPARC V8 TISA processor 712 to perform testing of SUT 720 while being controlled by HC 701.

As depicted in FIG. 7, there is an interface 709 between HC 701 and TS 710. The interface 709 may support local communications and/or remote

communications (e.g., via a network) between HC 701 and TS 710. Thus, HC 701 may control interactive testing of SUT 720 via TS 710 locally and/or remotely.

In the exemplary TISA-based testing environment 700, the HC 701 is able to control, step-by-step, test execution on SUT 720, by controlling the operation of TS 710 via interface 709, thereby enabling interactive testing and debugging capabilities.

It will be appreciated that most of the left-hand side of FIG. 7 reuses existing Computer Science elements: namely, the entire HC 701, as well as the GDB stub 715 on TS 710. It is the same for the central part of FIG. 7, where analogies between HC 701 and TS 710 (as well as their associated sub-elements) are evident. The TISA allows this entire infrastructure to be leveraged to provide system testing.

As an example, in reference to the system test environment 500 of FIG. 5A (including the associated exemplary C programs, SVF instructions, TISA assembler instructions, and TISA codings), there are many interactive test operations that the TISA can enable by leveraging on GDB (or any other suitable debuggers), such as: (a) step-by-step execution while monitoring the variables "sent_value" and "received_value"; (b) on-the-fly modification of the value to be sent to the tap (variable "sent_value"); (c) modification of the looping end condition; (d) monitoring of all variables; and the like, as well as various combinations thereof. These interactive test operations are standard operations for GDB, and the TISA can directly use them, due to the ability of the TISA to automatically hand of control between the algorithmic and test access portions, as described hereinabove. In the absence of the TISA, special tooling would need to be developed and adapted to each hand-off implementation.

Although exemplary TISA-based testing environment 700 of FIG. 7 is primarily depicted and described herein with respect to using the GNU Tool Suite to support interactive testing of a specific system under test, it will be

appreciated, by those skilled in the art and informed by the teachings herein, that interactive testing capabilities in a TISA-based test environment may be realized using any suitable tool suites for testing any type of system under test.

Although TISA-based testing environment 600 of FIG. 6 and exemplary TISA-based testing environment 700 of FIG. 7 are primarily depicted and described herein with respect to linear test procedures where testing is done step-by-step following a pre-determined algorithm (for purposes of clarity in describing the interactive testing capabilities that are enabled by TISA), it will be appreciated that other more complicated interactive testing scenarios are possible due to the leverage of Computer Science experience and techniques enabled by TISA. An example of a more complicated interactive testing scenario enabled by TISA is depicted and described herein with respect to FIG. 8. It will be appreciated that this is merely one example, and that one skilled in the art and informed by the teachings herein may use TISA in many other interactive testing scenarios and applications.

As described herein, in addition to supporting both granularity and interaction, the TISA also supports concurrency.

The TISA naturally and fully merges the system testing flow with the computer science software flow and, therefore, can leverage the best aspects of both flows. As an example, approaches such as STAPL have difficulty in handling concurrent control of instruments, because such approaches are, by definition, fully sequential. Furthermore, approaches such as the MTC and SystemBIST are intrinsically sequential and single-task and, thus, it would be difficult and awkward to program such approaches to support concurrency. By contrast, concurrent execution is a well-known problem in Computer Science and is now, for instance, at the base of all operating systems. A large number of libraries supporting concurrent execution are available (e.g., the POSIX suite, the BOOST suite, and the like), and most modern processors are designed to efficiently support multi-tasking and context-switching (e.g., the

SPARC V8, for instance, implements a rotating register window). The natural interaction between the system testing flow and the computer science software flow that is enabled by the TISA allows the TISA to completely leverage such computer science approaches to concurrency.

The support of concurrency capabilities by the TISA may be better understood by way of an example. As an example, consider the problem of optimizing the data transfer rate of the T-R channel between the transmitter 522 and the receiver 526 of the system under test 520 of FIG. 5A and FIG. 7. This would involve transmitting a stream of data patterns from transmitter 522 on first board 521, receiving a corresponding stream of data patterns at receiver 526 on second board 525, and comparing the transmitted and received streams of data patterns to compute bit/error rates and to tune parameters of transmitter 522 and/or receiver 526 accordingly. This optimization may be performed efficiently using three programs operating concurrently.

FIG. 8 depicts an exemplary program architecture for performing optimization of the transmitter-receiver channel of the system under test of FIG. 5A and FIG. 7.

As depicted in FIG. 8, exemplary program architecture includes a pattern generator 802, a pattern receiver 804, and a comparator 806. The pattern generator 802, pattern receiver 804, and comparator 806 cooperate to optimize the data transfer rate of the T-R channel between the transmitter 522 and the receiver 526 of the system under test 520 of FIG. 5A and FIG. 7.

The pattern generator 802 sends the appropriate input data patterns to the transmitter 522 (T) on first board 521. The pattern generator 802 can access the TAP (illustratively, TAP 510 in FIG. 5A, TAP 718 in FIG. 7) in order to provide the input data patterns to transmitter 522 via the scan chain 523 of first board 521 (B1). The pattern generator 802 may provide the input data patterns to the transmitter 522 in any suitable manner (e.g., as specified in lines 12-13 of the code described herein with respect to FIG. 5A). The input

data patterns may be any data patterns suitable for optimizing the T-R channel between transmitter 522 and receiver 526. For example, the input data patterns may be pre-computed patterns, random patterns, and the like, as well as various combinations thereof.

The pattern receiver 804 collects the appropriate output data patterns from the receiver 526 (R) on second board 525. The pattern receiver 804 can access the TAP (illustratively, TAP 510 in FIG. 5A, TAP 718 in FIG. 7) in order to collect the output data patterns from receiver 526 via the scan chain 527 of second board 525 (B2). The pattern receiver 804 may collect the output data patterns from the receiver 526 in any suitable manner (e.g., as specified in lines 14-15 of the code described herein with respect to FIG. 5A).

The comparator 806 communicates with pattern generator 802 and pattern receiver 804. The comparator compares the input data patterns and the output data patterns. The comparator 806 evaluates the bit transmission rate and the bit error rate of the T-R channel and, based on the results of the comparison, can access the control registers of both the transmitter 522 and the receiver 526 (omitted from FIG. 5A and FIG. 7, for purposes of clarity) to optimize the parameters of the T-R channel.

In order to perform such an optimization testing procedure, pattern generator 802, pattern receiver 804, and comparator 806 need to work in parallel, and each must be able to access the TAP independently of the others. This type of control structure is very difficult to code in traditional environments, which are developed only to support one-point serial handoff control over the TAP. This type of control structure also is very difficult to code in environments employing MTC or other such approaches which also share the same serial TAP access paradigm. By contrast, the TISA is not designed with any such assumption regarding test access; rather, in the TISA, test access is handled in a manner similar to other processor resources, and test access instructions are mixed directly with classical ISA instructions. Using the TISA, the optimization testing procedure of FIG. 8 may be executed by

any multitasking Operating System using standard constructs like processes, threads, inter-process communications (IPC), and the like, as well as various combinations thereof. In this manner, pattern generator 802, pattern receiver 804, and comparator 806 can share access to the TAP, and can resolve any eventual TAP sharing issues as is done for all processor resources, e.g., using well-known constructs and algorithms such as, for example, Dijkstra's semaphores. Thus, whereas existing system testing capabilities do not support concurrency, it is clear that the TISA easily and fully supports concurrency.

As described hereinabove, the TISA does not make any assumptions regarding the test access method or the associated test program partitioning; rather, test instructions are treated in the same manner, or substantially the same manner, as classical ISA instructions, without any a priori separation between the two. This enables the TISA to be completely compatible with all existing (and, most likely, future) computer science algorithms and constructs, something that no existing test processor approaches can support.

Thus, it will be appreciated that any existing software libraries can be ported into the TISA architecture. For example, it would be easy to obtain multitasking and concurrency (e.g., as depicted and described herein with respect to FIG. 8) by exploiting the POSIX and BOOST suites. Further, it will be appreciated that where the TISA is obtained as a generalization of an existing ISA (e.g., as depicted and described with respect to the exemplary SPARC V8 TISA implementation depicted and described with respect to the FIG. 5A and FIG. 5B), porting may not even be necessary since the ISA that the TISA has been developed from will already include such software libraries.

Furthermore, it will be appreciated that various other computer science techniques may be utilized for providing improved system testing using the TISA. For example, some examples of such computer science techniques which may be leveraged for the TISA include: (a) use of platform-independent

coding styles, (b) use of ISA-to-ISA converters; (c) use of a Virtual Machine approach, e.g., like for Java, to obtain platform-independent bytecode, or even extension of the Java Virtual Machine itself to become a TISA; and (d) use of an Application Programming Interface (API) to standardize some TISA software interfaces, which would then be translated into primitives by the appropriate drivers. It will be appreciated that these examples are merely a few examples of computer science techniques which may be leveraged for the TISA.

FIG. 9 depicts one embodiment of a method for adapting an Instruction Set Architecture (ISA) flow of a processor to form a Test Instruction Set Architecture (TISA) flow including TISA instructions adapted for use by the processor in testing at least a portion of a system under test.

Although primarily depicted and described herein as being performed serially, at least a portion of the steps of method 900 may be performed contemporaneously, or in a different order than depicted and described with respect to FIG. 9.

At step 902, method 900 begins.

At step 904, a first set of instructions is generated. The first set of instructions includes ISA instructions supported by the processor (i.e., ISA instructions being leveraged to provide the TISA for the processor).

At step 906, a second set of instructions is generated. The second set of instructions includes test instructions associated with the system under test. The second set of instructions may be generated in any suitable manner, e.g., as depicted and described with respect to TGT 210 of FIG. 2, as depicted and described with respect to TGT 310 of FIG. 3, and/or using any other suitable method of generating test instructions.

At step 908, the first set of instructions and the second set of instructions are integrated to form thereby TISA instructions. The TISA instructions provide the TISA for the processor.

At step 910, the TISA instructions are stored, displayed, propagated, and/or executed, or any combination thereof. The TISA instructions may be handled in any other suitable manner.

At step 912, method 900 ends.

The TISA may be formed in any suitable manner, e.g., as depicted and described with respect to method 1000 of FIG. 10, as depicted and described with respect to the test system of FIG. 2 and associated method 1110 of FIG. 11A, as depicted and described with respect to the test system of FIG. 3 and associated method 1120 of FIG. 11B, and/or using any other suitable method of forming a TISA.

FIG. 10 depicts one embodiment of a method for generating instructions adapted for use in testing at least a portion of a system under test. Although primarily depicted and described herein as being performed serially, at least a portion of the steps of method 1000 may be performed contemporaneously, or in a different order than depicted and described with respect to FIG. 10. At step 1002, method 1000 begins.

At step 1004, a first set of instructions is generated. The first set of instructions includes instructions generated by compiling at least one computer science software file (e.g., ISA instructions of an ISA supported by a processor).

At step 1006, a second set of instructions is generated. The second set of instructions includes test instructions generated by compiling at least one description file associated with the system under test.

At step 1008, the first and second sets of instructions are combined to form a combined set of instructions. In the combined set of instructions, the instructions of the first set of instructions are adapted for use in controlling execution of the test instructions of the second set of instructions.

At step 1010, the combined set of instructions is stored, displayed, propagated, and/or executed, or any combination thereof. The combined set of instructions may be handled in any other suitable manner.

At step 1012, method 1000 ends.

FIG. 11A and FIG. 11B depict more detailed embodiments of the method 900 depicted and described with respect to FIG. 9 and/or the method 1000 depicted and described with respect to FIG. 10.

FIG. 11A depicts one embodiment of a method for generating instructions adapted for use in testing at least a portion of a system under test. Although primarily depicted and described herein as being performed in a specific sequence, at least a portion of the steps of method 1110 of FIG. 11A may be performed in a different order than depicted and described with respect to FIG. 11A. FIG. 11A may be better understood by viewing it in conjunction with FIG. 2 and the associated description of FIG. 2.

At step 1111, method 1000 begins.

At step 1112, a program model is generated. The program model is generated by compiling at least one computer science software file (e.g., ISA instructions of an ISA supported by a processor), where the at least one computer science software file includes at least one call.

At step 1113, a first set of instructions is generated. The first set of instructions is generated using the program model. At least one computation request also is generated using the at least one call included in the at least one computer science software file.

At step 1114, a circuit model is generated. The circuit model is generated by compiling at least one system description file associated with the system under test.

At step 1115, a second set of instructions is generated. The second set of instruction is generated using the circuit model and the at least one computation request.

At step 1116, the first and second sets of instructions are combined to form a combined set of instructions. In the combined set of instructions, the instructions of the first set of instructions are adapted for use in controlling execution of the test instructions of the second set of instructions.

At step 1117, the combined set of instructions is stored, displayed, propagated, and/or executed, or any combination thereof. The combined set of instructions may be handled in any other suitable manner.

At step 1118, method 1000 ends. FIG. 11B depicts one embodiment of a method for generating instructions adapted for use in testing at least a portion of a system under test. Although primarily depicted and described herein as being performed serially, at least a portion of the steps of method 1120 of FIG. 11B may be performed contemporaneously, or in a different order than depicted and described with respect to FIG. 11B. FIG. 11B may be better understood by viewing it in conjunction with FIG. 3 and the associated description of FIG. 3.

At step 1121, method 1100 begins.

At step 1122, at least one pre-processed computer science software file and at least one test operation description file are generated by pre-processing at least one computer science software file.

At step 1123, a circuit model is generated. The circuit model is generated by compiling at least one system description file associated with the system under test and the at least one test operation description file.

At step 1124, a set of test operations is generated. The set of test operations is generated using the circuit model. The test operations from the set of test operations are described using a set of test primitives (e.g., test primitives defined by a test generation tool which generates the circuit model). The set of test primitives includes test operations adapted for use in testing the system under test.

At step 1125, the set of test operations is translated into a set of test instructions by translating the test primitives of the set of test operations into test instructions adapted for use in combination with software instructions of an instruction set architecture.

At step 1126, a program model is generated. The program model is generated by compiling the at least one pre-processed computer science software file and the set of test instructions.

At step 1127, a combined set of instructions is generated. The combined set of instructions is generated using the program model. The combined set of instructions includes (a) software instructions determined from the at least one pre-processed computer science software file and (b) test instructions from the set of test instructions.

At step 1128, the combined set of instructions is stored, displayed, propagated, and/or executed, or any combination thereof. The combined set of instructions may be handled in any other suitable manner.

At step 1129, method 1120 ends.

FIG. 12 depicts an exemplary embodiment of a TISA processor architecture.

As depicted in FIG. 12, TISA processor architecture 1200 includes a TISA processor 1210 and a memory 1220.

The TISA processor 1210 may be any processor that is suitable for performing system testing using a TISA, such as a SPARC V8 processor, an INTEL processor, or any other suitable processor.

The memory 1220 may include any memory suitable for use by TISA processor 1210 to support system testing using a TISA, including one or more of random access memory, persistent memory, and the like, as well as various combinations thereof. The memory 1220 may store any information required for performing system testing using a TISA, such as test programs, TISA instructions, testing data, and the like, as well as various combinations thereof.

In one embodiment, for example, TISA processor architecture 1200 of FIG. 12 may support the TISA flows depicted and described with respect to FIG. 2 and FIG. 3.

In one embodiment, for example, TISA processor architecture 1200 of FIG. 12 may operate in a manner similar to TISA processor 612 and memory 614 of testing system 610 depicted and described with respect to FIG. 6. For example, TISA processor architecture 1200 of FIG. 12 may be implemented using a SPARC V8 TISA processor and associated memory, such as in the testing system 710 depicted and described with respect to FIG. 7. In such an embodiment, the TISA processor 1210 itself interprets and executes both the ISA and TISA instructions.

In one embodiment, an apparatus for use in testing at least a portion of a system under test via a Test Access Port (TAP) includes a memory for storing a set of instructions of a test instruction set architecture and a processor executing the set of instructions of the test instruction set architecture for testing at least a portion of the system under test via the TAP. The set of instructions of the test instruction set architecture includes a first set of instructions comprising a plurality of instructions of an Instruction Set Architecture (ISA) supported by the processor and a second set of instructions comprising a plurality of test instructions associated with the TAP, where the instructions of the first class of instructions and the instructions of the second class of instructions are integrated to form thereby the set of instructions of the test instruction set architecture.

In one embodiment, a TISA processor for use in testing at least a portion of a system under test via a Test Access Port (TAP) includes a first class of instructions including instructions of an Instruction Set Architecture (ISA) supported by the processor and a second class of instructions including test instructions associated with the TAP, wherein the ISA instructions of the first set of instructions and the test instructions of the second set of instructions are integrated to form a TISA adapted for testing at least a portion of the system under test.

In one embodiment, a computer processor, for testing a system under test (SUT) via a Test Access Port (TAP), includes circuitry configured to

process instructions according to a test instruction set architecture (TISA) having semantics that enable interaction with the system under test via the TAP. The TISA includes a plurality of instructions of a first type and a plurality of instructions of a second type, where the first type of instructions include instructions of an instruction set architecture (ISA) supported by the computer processor and the second type of instructions include test instructions for testing the system under test via the TAP.

Although primarily depicted and described hereinabove with respect to embodiments in which the TISA processor is defined in a particular manner (e.g., using particular language to describe different classes and/or types of instructions), it will be appreciated that a TISA may be defined in other ways that are fully supported by the depiction and description of various TISAs as provided herein.

Although primarily depicted and described herein with respect to embodiments in which the TISA processor architecture is implemented using a single processor to support the TISA, in other embodiments the TISA processor architecture may be implemented using multiple processors.

FIG. 13 depicts an exemplary embodiment of a test processor architecture utilizing multiple processors to provide system testing capabilities.

As depicted in FIG. 13, test processor architecture 1300 includes a primary processor 1310 and a secondary processor 1320 in communication via a communication path 1330.

The primary processor 1310 may be any processor suitable for supporting system testing, such as a SPARC V8 processor, an INTEL processor, or any other suitable processor. The primary processor 1310 executes instructions for testing a system under test. In one embodiment, for example, primary processor 1310 may support testing functions similar to the functions supported by CPU 1210 of TISA processor architecture 1200 of FIG. 12 (e.g., where test processor architecture 1300 utilizes a TISA). In one embodiment, for example, primary processor 1310 may support testing

functions supported by testing processors in test processor architectures that do not utilize a TISA. The primary processor 1310 may support various other testing capabilities.

The secondary processor 1320 may be any processor suitable for supporting system testing, such as a SPARC V8 processor, an INTEL processor, or any other suitable processor. The secondary processor 1320 supports a Test Access Port (TAP) interface to the system under test (which is omitted for purposes of clarity). The TAP interface may interface with any suitable TAP. For example, the TAP interface may provide an interface to an IEEE 1149.1 TAP or any other suitable TAP which may be used for testing a system under test.

The primary processor 1310 and secondary processor 1320 cooperate to perform testing of at least a portion of a system under test.

The primary processor 1310 executes test instructions for testing a system under test. The test instructions may be test instructions of a TISA (where test processor architecture 1300 utilizes a TISA) or test instructions not associated with a TISA (where test processor architecture 1300 does not utilize a TISA). The primary processor 1310, during execution of the test instructions, detects instructions related to control of the TAP of the system under test (e.g., such as instructions for loading input data to a TAP controller of the system under test, instructions for reading output data from a TAP controller of the system under test, and like instructions, as well as various combinations thereof). The primary processor 1310 provides the TAP-related instructions to secondary processor 1320. The secondary processor 1320 receives the TAP-related instructions from primary processor 1310. The secondary processor 1320 executes the TAP-related instructions. The primary processor 1310 continues executing test instructions while secondary processor 1320 executes the TAP-related instructions received from primary processor 1310. In this manner, primary processor 1310 may perform a context switch and continue operating while secondary processor 1320

controls scan operations via the TAP of the system under test. This is difficult using a single-processor approach, because while the single processor is controlling the TAP, the single processor is prevented from performing other operations. Therefore, the use of multiple processors, as in the test processor architecture 1300, provides a significant improvement in testing efficiency without a need to use high-end processors, especially considering that operations over the TAP typically take a long time compared to the time required for a processor to perform a single operation.

The cooperation between primary processor 1310 and secondary processor 1320 to perform testing of at least a portion of a system under test is facilitated by communication path 1330. The communication path 1330 may be implemented using any suitable means of communication between primary processor 1310 and secondary processor 1320, which may depend on the type of multi-processor architecture with which the test processor architecture 1300 is implemented. For example, communication path 1330 may include one or more of a main processor interface bus, an auxiliary processor interface, a communication interface (e.g., such as a serializer-deserializer (SERDES) interface or other suitable communication interface), and the like, as well as various combinations thereof.

Although omitted for purposes of clarity, it will be appreciated that the test processor architecture 1300 will include memory (e.g., random access memory, persistent memory, cache memory, and the like, as well as various combinations thereof). The memory of test processor architecture 1300 may include one or more of memory shared by primary processor 1310 and secondary processor 1320, memory dedicated to primary processor 1310, memory dedicated to secondary processor 1320, and the like, as well as various combinations thereof.

Although omitted for purposes of clarity, it will be appreciated that the test processor architecture 1300 may include various other support circuits,

such as buses, I/O circuits, and the like, as well as various combinations thereof.

The test processor architecture 1300 of FIG. 13 may be implemented in a number of ways.

In one embodiment, for example, the test processor architecture may use a test co-processor unit architecture in which a central processor unit (CPU) cooperates with a test co-processor unit (TCPU) in order to support system testing. An exemplary embodiment is depicted and described with respect to FIG. 14.

In one embodiment, for example, the test processor architecture may use a test adjunct processor unit architecture in which a central processor unit (CPU) cooperates with a test adjunct processor unit (TAPU) in order to support system testing. An exemplary embodiment is depicted and described with respect to FIG. 15.

FIG. 14 depicts an exemplary embodiment of a test co-processor architecture. The test co-processor architecture 1400 is suitable for use as a TISA processor architecture for supporting system testing using a TISA. The test co-processor architecture 1400 also is suitable for use as a test processor architecture for supporting system testing that does not employ a TISA.

The test co-processor architecture 1400 includes a central processor unit (CPU) 1410, a test co-processor unit (TCPU) 1420, a main memory 1430, and a flash memory 1440.

The test co-processor architecture 1400 includes a main processor interface bus 1451. The CPU 1410, TCPU 1420, main memory 1430, and flash memory 1440 each are coupled to (or otherwise configured to be able to communicate with) the main processor interface bus 1451.

The test co-processor architecture 1400 also may include an auxiliary processor interface 1452 which directly couples CPU 1410 and TCPU 1420, thereby enabling direct communications between CPU 1410 and TCPU 1420.

The CPU 1410 may be any CPU suitable for performing system testing of a system under test. The CPU 1410 supports testing capabilities supported by primary processor 1310 depicted and described with respect to FIG. 13.

The TCPU 1420 may be any CPU suitable for facilitating system testing of a system under test. The TCPU 1420 supports a Test Access Port (TAP) interface 1460, which may interface with any suitable TAP (e.g., such as an IEEE 1149.1 TAP or any other suitable TAP used for testing a system under test). The TCPU 1420 supports testing capabilities supported by secondary processor 1320 depicted and described with respect to FIG. 13.

The CPU 1410 and TCPU 1420 cooperate to perform testing of at least a portion of a system under test in a manner similar to primary processor 1310 and secondary processor 1320 depicted and described with respect to FIG. 13. The CPU 1410 and TCPU 1420 utilize instruction exception handling in order to enable CPU 1410 to continue operating to process test instructions while TCPU 1420 executes TAP-related instructions for controlling the TAP of the system under test during testing.

The CPU 1410 executes test instructions for testing a system under test. The CPU 1410, during execution of the test instructions, detects instruction exceptions (i.e., instructions related to control of the TAP of the system under test) and provides the instruction exceptions to TCPU 1420. The TCPU 1420 receives the instruction exceptions from CPU 1410 and processes the instruction exceptions such that the TCPU 1420 may handle the instruction exceptions while CPU 1410 continues to operate to perform other tasks (e.g., executing other testing instructions). In other words, CPU 1410 and TCPU 1420 cooperate during system testing such that CPU 1410 may switch context and continue to operate to perform other tasks while TCPU 1420 handles instruction exceptions detected by CPU 1410, thereby improving system testing efficiency.

In one embodiment, the CPU 1410 includes a cache 1411, e.g., for improving the performance of CPU 1410.

In one embodiment, the TCPU 1420 includes a direct memory access (DMA) unit 1421, which may be any type of DMA unit suitable for use in support system testing. In one embodiment, for example, DMA unit 1421 is a scatter/gather (S/G) DMA unit. The TCPU 1420 may utilize DMA unit 1421 for purposes of handling instruction exceptions received from CPU 1410, and for efficiently accessing sensible data stored in memory. In one embodiment, CPU 1410 may configure S/G DMA tables prior to encountering an instruction exception.

In one embodiment, the TCPU 1420 supports a set of specialized TCPU instructions. The set of specialized TCPU instructions may support TAP access and control. The set of specialized TCPU instructions may be used by TCPU 1420 to perform specific TAP operations on the TAP State Machine.

The CPU 1410 and TCPU 1420 utilize main memory 1430 and/or flash memory 1440 for performing various testing functions, such as execution of test instructions by CPU 1410, instruction exception handling by TCPU 1420, execution of TCPU instruction by TCPU 1420, and the like, as well as various combinations thereof. The main memory 1430 may be any suitable processor memory. The flash memory 1440 may be any suitable flash memory or any other suitable form of persistent memory. The CPU 1410 and TCPU 1420 share the memory with arbitrated access. The CPU 1410 and TCPU 1420 also may share the memory for purposes of exchanging information. Although primarily depicted and described with respect to specific numbers and types of memory, it will be appreciated that various other memory schemes may be used for supporting the functions performed by CPU 1410 and TCPU 1420.

The CPU 1410 and TCPU 1420 perform testing of the system under test using communication between CPU 1410 and TCPU 1420 and communication between CPU 1410 and/or TCPU 1420 and other components of test co-processor architecture 1400 (e.g., main memory 1430, flash memory 1440, and other components), and the like, as well as various

combinations thereof. The communications may be supported using one or both of the main processor interface bus 1441 and the auxiliary processor interface 1452. The communications between CPU 1410 and TCPU 1420 may include communications associated with instruction exception notification, interrupt access, DMA arbitration, and the like, as well as various combinations thereof. The communications between CPU 1410 and TCPU 1420 and other components of the test co-processor architecture 1400 may include communications associated with reading from memory, writing to memory, and/or any other tasks which may be performed in support of testing the system under test.

FIG. 15 depicts an exemplary embodiment of a test adjunct processor architecture. The test adjunct processor architecture 1500 is suitable for use as a TISA processor architecture for supporting system testing using a TISA. The test adjunct processor architecture 1500 also is suitable for use as a test processor architecture for supporting system testing that does not employ a TISA.

The test adjunct processor architecture 1500 includes a central processor unit (CPU) 1510 and a test adjunct processor unit (TAPU) 1520. The CPU 1510 and TAPU 1520 may reside on the same board or may reside on different boards.

The CPU 1510 may be any CPU suitable for performing system testing of a system under test. The CPU 1510 supports testing capabilities supported by primary processor 1310 depicted and described with respect to FIG. 13.

The CPU 1510 has a main memory 1530M, a flash memory 1530F, and an input/output module 1540 associated therewith. The CPU 1510 has a main processor interface bus 1550 associated therewith. The CPU 1510, main memory 1530M, flash memory 1530F, and input/output module 1540 each are coupled to (or otherwise configured to be able to communicate with) the main processor interface bus 1550.

In one embodiment, the CPU 1510 includes a cache 1511, e.g., for improving the performance of CPU 1510.

The TAPU 1520 may be any CPU suitable for facilitating system testing of a system under test. The TAPU 1520 includes an input/output module 1521. The TAPU 1520 supports a Test Access Port (TAP) interface 1590, which may interface with any suitable TAP (e.g., such as an IEEE 1149.1 TAP or any other suitable TAP used for testing a system under test). The TAPU 1520 supports testing capabilities supported by secondary processor 1320 depicted and described with respect to FIG. 13.

The TAPU 1520 has a local test memory 1560 associated therewith. The TAPU 1520 has an internal interface bus 1570 associated therewith. The TAPU 1520 and local test memory 1560 each are coupled to (or otherwise configured to be able to communicate with) the internal interface bus 1570.

The input/output module 1540 associated with CPU 1510 and the input/output module 1521 of TAPU 1520 support a communication interface 1580 enabling communications between CPU 1510 and TAPU 1520. The communication interface 1580 supports streaming of TAP-related commands from CPU 1510 to TAPU 1520.

In one embodiment, the input/output module 1540 associated with CPU 1510 and the input/output module 1521 of TAPU 1520 support Serializer-Deserializer (SERDES) communications capabilities and, therefore, the communications interface 1580 is a SERDES-based communications interface. In this embodiment, the SERDES-based communications interface 1580 may be implemented using any suitable SERDES communications protocol (e.g., such as Gigabit Ethernet (GigE), Serial Rapid IO (SRIO), Peripheral Component Interconnect Express (PCIe), and the like). Although primarily depicted and described herein with respect to using SERDES-based communications between the CPU 1510 and the TAPU 1520, other suitable communications capabilities may be used in order to support communications between CPU 1510 and TAPU 1520.

The CPU 1510 and TAPU 1520 cooperate to perform testing of at least a portion of a system under test in a manner similar to primary processor 1310 and secondary processor 1320 depicted and described with respect to FIG. 13. The CPU 1510 and TAPU 1520 utilize command streaming via the communication interface 1580 in order to enable CPU 1510 to continue operating to process test instructions while TAPU 1520 executes TAP-related instructions for controlling the TAP of the system under test during testing.

The CPU 1510 executes test instructions for testing a system under test. The CPU 1510, during execution of the test instructions, detects instructions related to control of the TAP of the system under test. The CPU 1510 propagates the TAP-related instructions to the TAPU 1520 via the communication interface 1580 (i.e., from CPU 1510 to input/output module 1540 via the main processor interface bus 1550, for propagation via communication interface 1580). The TAPU 1520 receives the TAP-related instructions from CPU 1510 and processes the TAP-related instructions such that the TAPU 1520 may handle control of the TAP while CPU 1510 continues to operate to perform other tasks (e.g., executing other testing instructions). In other words, CPU 1510 and TAPU 1520 cooperate during system testing such that CPU 1510 may switch context and continue to operate to perform other tasks while TAPU 1520 handles TAP-related instructions detected by CPU 1510, thereby improving system testing efficiency.

In one embodiment, the TAP-related instructions detected by CPU 1510 and processed by TAPU 1520 are packetized by the CPU 1510 for propagation to TAPU 1520.

In one embodiment, the TAP-related instructions detected by CPU 1510 and processed by TAPU 1520 include opcodes supported by TAPU 1520. In one such embodiment, the TAP-related instructions also may include one or more extension commands adapted for use in performing block memory copies between memory associated with the CPU 1510 and memory

associated with the TAPU 1520 (e.g., between main memory 1530M and local test memory 1560).

The CPU 1510 utilizes main memory 1530M and/or flash memory 1530F for performing various testing functions, such as execution of test instructions, detection of TAP-related instructions, packetization of TAP-related instructions, and the like, as well as various combinations thereof. The main memory 1530M may be any suitable processor memory. The flash memory 1530F may be any suitable flash memory or any other suitable persistent memory.

The TAPU 1520 utilizes local test memory 1560 for performing various testing functions, such as storage of TAP-related instructions received from CPU 1510, processing of TAP-related instructions received from CPU 1510, and the like, as well as various combinations thereof. The local test memory 1560 may be any suitable processor memory. In one embodiment, the local test memory 1560 may be relatively small since it handles processing of scan chain segments of the scan chain of the system under test, rather than the entire scan chain (as may be required in an on-chip memory).

Although primarily depicted and described with respect to specific numbers and types of memory, it will be appreciated that various other memory schemes may be used for supporting the functions performed by CPU 1510 and TCPU 1520.

Although primarily depicted and described herein with respect to use of a co-processor architecture or an adjunct processor architecture to implement the TISA, it will be appreciated that the TISA may be implemented using any suitable processor architecture, which may include processor architectures other than the co-processor architecture or the adjunct processor architecture. Thus, the TISA processor architecture may be implemented using multiple processors in various other ways, at least some of which may include use of more than two processors for supporting the TISA.

Although primarily depicted and described herein with respect to use of the co-processor architecture or the adjunct processor architecture in order to implement the TISA architecture, it will be appreciated by one skilled in the art and informed by the teachings herein that the co-processor architecture and the adjunct processor architecture each may be used to implement other types of testing architectures (i.e., other testing architectures that do not employ TISA).

It will be appreciated that the test co-processor architecture and the test adjunct processor architecture are functionally similar in that each enables a TISA to be executed by two communicating processors. In a given application, the choice between the two architecture may be made by the designer on the basis of implementation-dependent parameters, such as available resources, costs, performances, physical constraints (integration in the same chip, in different chips and/or boards or any combination thereof), as well as any other implementation parameter. Although primarily depicted and described herein with respect to test co-processor and test adjunct processor architectures, it will be appreciated by one skilled in the art and informed by the teachings herein that these implementation considerations will apply to any other types of testing architectures/infrastructure.

The TISA processor architectures depicted and described herein may employ any suitable TISA for use in performing system testing.

A description of one exemplary embodiment of a TISA adapted for use with the TISA processor architectures follows. This exemplary embodiment of a TISA is an implementation of Scan Segment Level primitives depicted and described herein. In a Scan Segment Level abstraction level, the overall scan chain of the system-under-test is divided into segments, which are then used as the data atom of the algorithm. It will be appreciated that the system-under-test may be partitioned into the scan segments by the algorithm developer, which may be a human and/or an automated tool. A more general description of the use of TISA to enable scan operations to be performed at the Scan

Segment Level, i.e., a description that is independent of this exemplary TISA implementation, is provided detail hereinbelow.

The following embodiment of a TISA proposes a set of registers and instructions able to define and handle those scan segments. The following embodiment is based on a 32-bit sized TISA, but it could be adapted to any other word size (e.g., 16-bit, 64-bit, or any other suitable word size).

FIG. 16 depicts an exemplary register set that can be used by a TISA processor. The exemplary TISA includes the four register sets (denoted as register sets R1 through R4), which are depicted in FIGs. 16A - 16D, respectively.

As depicted in FIG. 16A, the first register set R1 includes the following User Accessible Data Registers:

- StatusRegister: 32-bit register containing status state information;
- Control Register: 32-bit register containing command encodings;
- BlockRegister: 32-bit register containing the offset in memory to Preformatted data structures indirectly pointing to the scan data in (gather data) and where to write the data out (scatter data) [Used for all scan and compare operations for accessing Scatter/Gather Segment Descriptions];
- ScanLengthRegister: 32-bit register where the current number of bits remaining to be scanned resides (also automatically populated from Scatter/Gather Segment Descriptions for block mode opcodes);
- ScanStateRegister: 32-bit register containing 3 banks of 4 bits representing the startState, scanState, and endState of a scan operation. The 4 bits represent the encoding of the 16 states of the TAP state machine. (also populated from Scatter/Gather Segment Descriptions in block mode); and
- UserDataRegisters[1 -1 1]: 32-bit registers containing scan segment data for small scan operations and data reuse (may be source or destination register).

As depicted in FIG. 16B, the second register set R2 includes the following Internal Scratch Registers:

- BlockPointerRegister: 32-bit register pointing to the current Scatter/Gather Segment Description reference to be processed during Multiple Scan Instructions;
- BlockCountRegister: 32-bit register containing the count of Scatter/Gather Segment Descriptions to be processed during Multiple Scan Instructions; and
- InstructionRegister: 32-bit register where the current opcode is placed for decoding.

As depicted in FIG. 16C, the third register set R3 includes the following Scatter/Gather Segment Descriptions registers:

- BlockOffsetField: 32-bit number describing the bank of address when 64-bit architectures are used;
- ScanLengthField: 32-bit integer specifying the number of bits to scan for this segment;
- StateTraversal Field: 3 fields of 4 bits each that represent the start state, scan state, and end state for this scan operation (each 4 bits represent the 16 state TAP State Machine states);
- SourceLocationField: 32-bit base address for where the TDI data resides in memory;
- DestinationLocationField: 32-bit base address for where the TDO data will be stored in memory;
- ExpectedValueField: 32-bit address for where the expected vector resides in memory;
- ResponseLocation Field: 32-bit base address for where the captured TDI data resides in memory;
- MaskField: 32-bit base address for where the MASK data used to limit the comparison operation resides in memory;

- ResultLocationField: 32-bit base address for where the results of the comparison will be stored in memory.

As depicted in FIG. 16D, the fourth register set R4 includes the following MultiBlock Scatter/Gather Segment Descriptions registers:

- BlockOffsetField: 32-bit number describing the bank of address when 64-bit architectures are used;
- BlockCountField: 32-bit number defining the number of scan segments that are represented by this MultiBlock scan (used to initialize the BlockCountRegister during a MultiBlock scan operation);
- ScatterGatherOpcodeField: 32-bit command opcode used for the Scatter/Gather Segment Description pointed to by the associated ScatterGatherBlockField; and
- ScatterGatherBlockField: 32-bit address for where the Scatter/Gather Segment Description associated with the previous ScatterGatherOpcodeField is located in memory.

It will be appreciated that the exemplary TISA register sets may be modified in any suitable manner. For example, each of the exemplary register sets may be modified to include fewer, more, and/or different registers. For example, the exemplary registers may be regrouped into fewer, more, and/or different sets. For example, fewer, more, and/or different register sets may be used. In other words, the exemplary TISA register sets may be replaced with any other TISA register set(s) suitable for use with TISA instructions sets to implement a TISA processor architecture.

The exemplary TISA may employ any suitable TISA instruction set (i.e., command dictionary) for use in performing system testing.

The exemplary TISA instruction set includes the following opcodes, which may be utilized to manipulate register sets R1 through R depicted and described with respect to FIGs. 16A - 16D, as well as the original ISA register sets depicted and described herein:

StateTransition <TMS Value>, <TCK cycles>

- This opcode is used to traverse the TAP state machine using the value of TMS for the given number of TCK clock cycles. This opcode is used to perform general state transitions between states of the TAP state machine. The <TMS Value> represents a single bit, while the <TCK cycles> represents the remaining data bits of the opcode.

RunTest <startState>, <testState>, <endState>

- This opcode is used to transition from <startState> to <testState>, and to loop in <testState> for the number of TCK cycles as specified by the ScanLengthRegister. This opcode is used to transition to the <endState> as the conclusion of looping.

Scan Register <source registers destination register>[,<expected register>] [,<mask register>]

- This opcode is used to scan the data in the user data register <source register> and store the captured value into the user data register destination registers If the <expected_register> is present, compare captured data with it and raise error accordingly, eventually using the <mask_register>, if present. The number of bits scanned is defined in the ScanLengthRegister (0 <= n < 32). The start, scan, and end states are defined in the ScanStateRegister.

Scan RegisterZero destination register>[,<expected register>] [,<mask register>]

- This opcode is used to scan the vector value of all "0" and store the captured value into the user data register destination registers The number of bits scanned is defined in ScanLengthRegister (0 <= n < 32). The start, scan, and end states are defined in the ScanStateRegister.

<expected_register> and <mask_register> are used as in the ScanRegister instruction.

ScanRegisterOne destination register>[,<expected register>] [,<mask register>]

- This opcode is used to scan the vector value of all "1" and store the captured value into the user data register destination registers The number of bits scanned is defined in ScanLengthRegister (0 <= n < 32). The start, scan, and end states are defined in the ScanStateRegister. <expected_register> and <mask_register> are used as in the ScanRegister instruction.

ScanBlock

- This opcode is used to scan the data pointed to by the BlockRegister to the SUT starting at the <startState>, scanning the data in the <scanState>, with the <endState> finalizing the operation state as defined by the Block's StateTraversal Field. The ScanStateRegister is populated with the data from the StateTraversal Field prior to the scan operation. The ScanLengthRegister is populated with the data from the ScanLengthField prior to the scan operation. No data from TDO is preserved. If the ExpectedValueField and Maskfield are set, comparison and error generation are done accordingly.

ScanBlockCapture

- This opcode is used to scan the data pointed to by the BlockRegister to the SUT starting at the <startState>, scanning the data in the <scanState>, with the <endState> finalizing the operation state as defined by the Block's StateTraversal Field. The ScanStateRegister is populated with the data from the StateTraversal Field prior to the scan operation. The ScanLengthRegister is populated with the data from the

ScanLengthField prior to the scan operation. The data captured from TDO is preserved. If the ExpectedValueField and Maskfield are set, comparison and error generation are done accordingly.

ScanBlockZeroCapture

- This opcode is used to scan the data vector of all "0" to the SUT starting at the <startState>, scanning the data in the <scanState>, with the <endState> finalizing the operation state as defined by the Block's StateTraversal Field capturing the result in the register defined to by the BlockRegister. The ScanStateRegister is populated with the data from the StateTraversal Field prior to the scan operation. The ScanLengthRegister is populated with the data from the ScanLengthField prior to the scan operation. If the ExpectedValueField and Maskfield are set, comparison and error generation are done accordingly.

ScanBlockZero

- This opcode is used to scan the data vector of all "0" to the SUT starting at the <startState>, scanning the data in the <scanState>, with the <endState> finalizing the operation state as defined by the Block's StateTraversal Field without capturing the result. The ScanStateRegister is populated with the data from the StateTraversal Field prior to the scan operation. The ScanLengthRegister is populated with the data from the ScanLengthField prior to the scan operation. If the ExpectedValueField and Maskfield are set, comparison and error generation are done accordingly.

ScanBlockOneCapture

- This opcode is used to scan the data vector of all "1" to the SUT starting at the <startState>, scanning the data in the <scanState>, with the <endState> finalizing the operation state

as defined by the Block's StateTraversal Field capturing the result in the register defined to by the BlockRegister. The ScanStateRegister is populated with the data from the StateTraversal Field prior to the scan operation. The ScanLengthRegister is populated with the data from the ScanLengthField prior to the scan operation. If the ExpectedValueField and Maskfield are set, comparison and error generation are done accordingly.

ScanBlockOne

- This opcode is used to scan the data vector of all "1" to the SUT starting at the <startState>, scanning the data in the <scanState>, with the <endState> finalizing the operation state as defined by the Block's StateTraversal Field without capturing the result. The ScanStateRegister is populated with the data from the StateTraversal Field prior to the scan operation. The ScanLengthRegister is populated with the data from the ScanLengthField prior to the scan operation. If the ExpectedValueField and Maskfield are set, comparison and error generation are done accordingly.

The exemplary TISA instruction set includes the following register modification instructions that use explicit values:

LoadRegisterExplicit <const value>, <register name>

- This instruction loads the constant value of <const value> into the register named by <register name>.

CopyRegister <source register>, destination register>

- This instruction copies the contents of the register named as <source register> into the register named by destination registers

The exemplary TISA instruction set includes the following register modification instruction that use implicit values:

LoadRegisterImplicit <user data register>, <register name>

- This instruction uses the value in the named <user data register> as a pointer reference to a memory location where the real data resides and stores the referenced value into the register named by <register name>

The exemplary TISA instruction set includes the following register preservation instructions:

StoreRegisterImplicit <register name>, <user data register>

- This instruction uses the value in the named <user data register> as a pointer reference to a memory location where the value in the register named by <register name> is to be stored.

StoreRegisterExplicit <register name>, <const value>

- This instruction stores the value of register named by <register name> into the memory location specified by <const value>.

The exemplary TISA instruction set includes the following logical operations on registers:

AND <source registers destination register>

- This operation performs a logical AND operation between the <source register> and the destination register> and places the resulting value in the destination registers

OR <source registers destination register>

- This operation performs a logical OR operation between the <source register> and the destination register> and places the resulting value in the destination registers

XOR <source registers destination register>

- This operation performs a logical XOR operation between the <source register> and the destination register> and places the resulting value in the destination registers

NOT <source register>, destination register>

- This operation performs a logical NOT operation on the <source register> and places the resulting value in the destination registers

XORM <source registers <mask register>, destination register>

- This operation performs a logical XOR operation between the user data register <source register> and the user data register destination registers comparing only those bits aligning with the user data register <mask register> bit containing a value of "1", and places the resulting value in the destination registers

  Note that uncompared bits will result in a "0" value in the destination registers

The exemplary TISA instruction set includes the following miscellaneous operation on registers:

NOP

- A no operation opcode to be used as a filler to provide alignment in some ISA instruction sets.

The exemplary TISA instruction set includes the following instructions for extending support for streaming for an embodiment using the adjunct processor architecture:

MemoryWrite

- This instruction writes to the local test memory using the following arguments: <sequence number>, <block offset (32-bit offset)>, <number of bytes to transfers destination address (in specified memory block)>, data byte(s)>.

MemoryRead

- This instruction reads from the local test memory using the following arguments: <sequence number>, <block offset (32-bit offset)>, <number of bytes to transfers <source address (in specified memory block)>. This instruction returns a stream of

data bytes tagged with the sequence number and the number of bytes being transferred.

The exemplary TISA instruction set includes the following values for scan state:

StartState, ScanState, EndState

- The scan state codes include: TestLogicReset (TLR), RunTest/Idle (RTI), PauseDR (PDR), PauseIR (PIR), ScanDR (SDR), ScanIR (SIR). There is a 4-bit representation per state code, and 12 bits are used to describe the entire state transition sequence for a scan operation.

It will be appreciated, by one skilled in the art and informed by the teachings herein, that various other TISA implementations may be used with the TISA processor architectures depicted and described herein. For example, other TISA implementations may use fewer, more, and/or different registers, may use fewer, more, and/or different instruction sets, and the like, as well as various combinations thereof. In one embodiment, other TISA implementations may be utilized where different processor architectures are used, in order to provide TISA implementations better-suited to specific applications, and/or for any other suitable reasons.

As described hereinabove, use of TISA in a JTAG architecture enables scan operations to be performed at the Scan Segments Level, which allows the definition of independently controllable "scan segments" inside the overall scan path, thereby providing a flexible and powerful set of primitives that can be used to define scan operations directly in the problem space and resolve the scan operations at implementation time.

In general, JTAG operations are based on the scan operation in which all bits are scanned in serially one-by-one while at the same time bits are being scanned out serially one-by-one. This means that, in order to be able to perform a scan operation, knowledge of which value is needed for each bit in the scan chain (i.e., the input and output vectors) is required. TGTs typically

provide this capability for traditional structural testing by computing the required vectors from a system model obtained through description languages such as BSDL. Additionally, formats like SVF and STAPL mirror this, as they allow the user to manipulate those vectors. While testing in this manner is sufficient for structural (and other types) of testing, testing in this manner is highly inefficient for interactive setups in which there is no real need to access the whole scan chain. The inefficiency may be seen by considering an example.

For example, consider a scan chain composed of 100 instruments, each one having 16 bits. If the user needed to write 0x1 234 in the registers of the 76th instrument in the scan chain, the TGT would need to generate the vector for the whole scan chain (100*16=1 600 bits) and send it to the TAP interface to be input into the scan chain. Similarly, if the user wanted to read the associated output, the TGT would need to receive the entire 1600 bit vector before being able to extract the desired output information. In this example, the fact that a majority of the scan bits are useless is not important, as scan efficiency is not one of the goals (rather, in this example, the goal is primarily to be able to efficiently access one particular entity of the scan chain).

This type of approach is a problem at least for the following reasons: (a) there is the computational need of handling long vectors (e.g., lots of memory transfers have a high impact on performances); (b) there is a need to store the entire vector(s) in memory (which may be a problem for long chains); (c) memory storage is not limited to data inputs and data outputs, but also includes expected data, input and output mask, and the like (thereby multiplying memory requirements which are already potentially strained just from the input and output data); and (d) the transformation from instrument-vector-instrument must be made each time (which demands computational power and time).

The Scan Segments Level abstraction level is a powerful tool for providing efficient access to individual entities, or groups of entities, of the scan chain of a system under test, without any special emphasis on scan efficiency (even if, of course, still enabling it if needed).

In one embodiment, Scan Segments Level abstraction is implemented by decomposing a scan chain into a succession of segments and defining one or more scan operations on each of the segments. The scan chain is composed of a plurality of elements, and each segment includes at least one of the elements of the scan chain. The elements may be defined at many levels of the system under test (e.g., elements may be devices, instruments, registers, segments of a register, and the like, as well as various combinations thereof), and, thus, that the segments into which the scan chain is decomposed may be defined at many levels of the system under test (e.g., segments may include one or more devices, a portion of a device(s), one or more instruments, a portion of an instrument(s), one or more registers, a portion of a register(s), one or more register segments, and the like, as well as various combinations thereof). In this manner, a segment may represent the smallest control unit of the scan chain.

In one embodiment, decomposition of a scan chain into segments may be hierarchical. For example, the scan chain may be decomposed into segments, at least some of which may be composed by sub-segments, at least some of which may be composed by sub-segments, and so forth. In this manner, the hierarchical decomposition of the scan chain may be viewed as having a tree-based structure in which a segment may be composed of other segments. In one such embodiment, the segments at the leaves of the tree may be referred to as segments (in that they represent the smallest control unit of the scan chain), which the segments located above the leaves of the tree may be referred to as super-segments. It will be appreciated that, in one embodiment, one or more of the segments of the scan chain may be composed of virtual sub-segments which are controllable, but only in a

manner that is transparent to the user/system). The hierarchical decomposition of a scan chain may be defined in any other suitable manner.

The use of segmentation enables definition of entities for types of segments and/or types of segment combinations. An entity is a generic description of a type of target, which is valid for and may be reused for each physical instance of that type of target. For example, an entity may define a description of a device, a group of devices, a portion of a device, an instrument, a group of instruments, a portion of an instrument, and the like, as well as various combinations thereof. Thus, since a scan chain may be decomposed such that segments of the scan chain include specific elements or combinations of elements, entities may be defined for respective segments and/or respective combinations of segments, of a scan chain. For example, where a scan chain is decomposed such that a segment includes an instrument, an entity may be defined for that type of segment (i.e., each segment including that type of instrument), such that the entity may be reused for each physical instance of that type of segment in the scan chain. Similarly, for example, where a scan chain is decomposed such that a segment includes multiple instruments, an entity may be defined for that type of segment (i.e., each segment including that type combination of instruments), such that the entity may be reused for each physical instance of that type of segment in the scan chain. This enables additional features and functions to be supported, as described below.

The use of segmentation allows an entity (i.e., a description of a type of segment under control) to be correlated with a physical protocol that is used to communicate with the entity. As a result, description languages (e.g., such as NSDL, P1687 IJTAG PDL, and the like) could be written specifically for the entity, and the connectivity description portion (e.g., the structure of the NSDL or the IJTAG HDL) would describe the ordering of the segmentation instructions.

TISA provides a reusable modularity that can be defined once for all occurrences of a particular entity type, as the TISA instructions are segment-based operations rather than model-based operations. Thus, since TISA is both modular and autonomous for the entity under test in a particular segment, TISA provides significant advantages over existing architectures.

TISA enables a direct mapping of the Test Data Register definition into a reusable and portable module that may be plugged into the execution flow at any point in the scan process, in any order that is necessary, without needing to define the entire connectivity as a static model up front as existing tools require. TISA enables integration of the port/signal interfaces that are non-scan with the scan operations as a single solution space architecture based on a unified control flow and standard computer science techniques (providing significant advantages over solutions in which native language constructs are used to provide access to non-scan operations).

TISA enables reuse of instruction sequences for multiple instances of the same entity, thereby enabling a reduction in code storage requirements in the system. For example, a generalized function, which maps to description language functions which are called by a managing program, may be written. In this example, each of the functions are methods of native language objects that represent the entity, and there may be separate instances of these objects for each entity defined in the system, but there could be a single copy of code used to communicate with each of these instances. In this manner, the native language implementation models directly control the description language used to specify the connectivity and functionality of the circuit.

In reference to the example given above, use of Scan Segments Level abstraction would enable definition of three segments as follows: segment S1 including instruments 1 through 75, segment S2 including the instrument 76, and segment S3 including instruments 77 through 100. In this manner, access to instrument 76 is greatly simplified. For example, access to instrument 76 could be obtained by making a "dummy shift" (e.g., ScanBlockZero) for

segment S3, executing the instruction(s) for segment S2 (i.e., instrument 76), making another dummy shift for segment S1, and then terminating with an update. In such a sequence, access to segment S2 (i.e., to a specific instrument in the scan chain) is provided without a need of any knowledge of segment S1 or segment S3 apart from their length. It will be appreciated that this is merely one example, and, thus, that other decompositions of the 100 instrument-long chains are possible to enable access to other instruments or instrument groups.

FIG. 17 depicts a high-level block diagram of a system under test, illustrating an exemplary decomposition of an exemplary scan chain of the system under test.

The exemplary SUT 1700 includes four devices 1710i - 1710_4 (collectively, devices 1710; and denoted in FIG. 17 as Device 1, Device 2, Device 3, and Device 4, respectively). The devices 1710 are arranged serially within SUT 1700 to form a scan chain 1720. The scan chain 1720 is as follows: the TDI of the TAP is connected to the input port of device 1710_4, the output port of device 1710_4 is connected to the input port of device 1710_3, the output port of device 1710_3 is connected to the input port of device 1710_2, the output port of device 1710_2 is connected to the input port of device 1710_1, and the output port of device 1710_1 is connected to the TDO of the TAP.

In the exemplary SUT 1700, each of the devices 1710 includes (1) an input de-multiplexer providing inputs to a test instruction register (TIR) and a test data register (TDR), and (2) an output multiplexer for collecting outputs from the TIR and the TDR. The TIR and TDR of each device 1710 are parallel registers. The device 1710_3 includes one additional TDR, such that the input de-multiplexer provides inputs to one TIR and two TDRs and collects outputs from the one TIR and the two TDRs, where the one TIR and two TDRs are all in parallel. The TIRs and TDRs each are depicted as serial shift registers, each including nine associated elements (e.g., flip-flops). In this manner, (a) the TIRs form one scan chain (denoted as an test instruction scan chain) that

includes thirty-six serial elements and (b) the TDRs form another scan chain (denoted as a test data scan chain) that includes forty-five total elements and thirty-six serial elements (i.e., because the two TDRs of device 1710_3 are in parallel).

In the exemplary SUT 1700, the test instruction scan chain has been decomposed into four segments follows: a first segment SI4 which includes the nine elements of the TIR of device 1710_4, a second segment SI3 which includes the nine elements of the TIR of device 1710_3, a third segment SI2 which includes the nine elements of the TIR of device 1710_2, and a fourth segment SI1 which includes the nine elements of the TIR of device 1710_1. In this manner, the testing system may access any of the TIRs of SUT 1700, individually or in combination, with minimal knowledge of the other TIRs of SUT 1700 (other than the number of elements of which they are composed).

In the exemplary SUT 1700, the test data scan chain has been decomposed into six serial segments (seven total segments) as follows: a first segment SD4 that includes the nine elements of the TDR of device 1710_4; a second segment SD3 that includes the nine elements of the TDR of device 1710_3; a third segment SD2 that includes either the nine elements of the first TDR of device 1710_2 (denoted as sub-segment SD2.1) or the nine elements of the second TDR of device 1710_2 (denoted as sub-segment SD2.2), where these are counted as separate segments for purposes of counting the total number of segments); and a fourth segment which is further decomposed into three serial sub-segments as follows: a first sub-segment that includes the first three elements of the TDR of device 1710i (denoted as sub-segment SD1.1), a second sub-segment that includes the next four elements of the TDR of device 1710_1 (denoted as sub-segment SD1.2), and a third sub-segment that includes the last two elements of the TDR of device 1710i (denoted as sub-segment SD1.3). In this manner, the testing system may access any of the TDRs of SUT 1700 (or even sub-portions of the TDR of device 1710i), individually or in combination, with minimal knowledge of the

other TDRs of SUT 1700 (other than the number of elements of which they are composed).

It will be appreciated that SUT 1700 of FIG. 17 is merely one example of the manner in which the scan chain(s) of a system under test may be decomposed for use in providing Scan Segments Level abstraction. Although depicted and described herein with respect to specific types, numbers, and arrangements of elements, components, and the like, it will be appreciated that a system under test for which a scan chain(s) is decomposed may be include various other types, numbers, and/or arrangements of elements, components, and the like.

As described herein, decomposition of the scan chain of a system under test enables scan operations to be defined on the segments, thereby improving testing efficiency. A method, according to one embodiment, for generating a set of instructions including scan operations for segments of a decomposed scan chain is depicted and described herein with respect to FIG. 18.

A more detailed example of scan decomposition and generation of scan segment operations is provided follows.

As a general example, consider a scan chain that includes three boards where each board includes a segment (denoted as segments A, B, and C associated with a first board, a second board, and a third board, respectively). In this example, where the scan segments are hierarchical, the segment A on the first board may be composed of a plurality of sub-segments (e.g., sub-segments A_1 through A_n), the segment B on the second board may be composed of a plurality of sub-segments (e.g., sub-segments B_1 through B_n), and/or the segment C on the third board may be composed of a plurality of sub-segments (e.g., sub-segments Ci through C_n).

As a more specific example, following the application and the SUT, a segment could be: one or more registers inside an instrument, an instrument,

a cluster ot registers, one or more boards, and the like, as well as various combinations thereof.

The overall scan operation is therefore decomposed in a series of segment scan operations. As a result, all that is required in order to obtain the final scan chain operation is a series of simple atomic operations. Thus, the embodiments of Scan Segments Level abstraction, while not exclusively limited to, are especially effective in implementations in which the atomic test operations are treated like processor operations (e.g., such as in the various TISA implementations depicted and described herein, or in any other similar implementations in which atomic test operations are treated like processor operations).

In such embodiments of Scan Segments Level abstraction, the actual implementation of the Scan Segments Level scan operations may require that one or more technological constraints linked to JTAG be addressed. For example, constraints such as the need to define the state of the TAP machine and the risk of using the Pause-DR state (not always implemented), among others, may need to be addressed.

In order to identify instrument/segment outputs in the output bitstream received via the scan chain, based on the position of the instrument/segment in the scan chain, the scan chain may be treated as a first-in-first-out (FIFO) system (given its serial nature) such that the first segment that is scanned in is also the first segment that is scanned out (as it is closest to the end of the scan chain).

In order to make the SUT "experience" the sequence of scan segment operations like a single scan operation, the TCK may be frozen between segment operations. As all elements inside the scan chain are synchronous, the effect of freezing TCK in this manner is that the scan chain is frozen together with TCK.

The use of Scan Segments Level in a TISA-based testing system may be better understood by way of a few examples, In the examples that follow,

assume that a system under test (SUT) is composed of three segments (denoted as A, B, and C, in that order), and that a user needs to write a value V inside of segment B.

As a first example, assume that the three segments of the system (A, B, and C) are implemented inside the same JTAG device. In this first example, once the three segments are defined in memory, the TISA operations would become:

i. Set Startstate = Run-Test-Idle, Scanstate=Endstate=ShiftDR;
ii. Set ScanLenghtField to the length of Segment A;
iii. Scan a bypass sequence into segment A;
iv. Set Startstate = Scanstate=Endstate=ShiftDR;
v. Set ScanLenghtField to the length of Segment B;
vi. Scan V into segment B;
vii. Set Startstate = Scanstate= ShiftDR, Endstate=Run-Test-Idle;
viii. Set ScanLenghtField to the length of Segment C;
ix. Scan a bypass sequence into segment C.

With respect to the first example, keeping the TAP Finite State Machine (FSM) in the ShiftDR state ensures that there is no update on the scan chain. This may be seen from the first example, in which keeping the TAP FSM in the ShiftDR state from step (i) to step (ix) ensures that there is no update on the scan chain, given that the UpdateDR State will be reached only once leaving ShiftDr.

Further with respect to the first example, the scan clock TCK is active only during the scan operations (i.e., steps (iii), (vi), and (ix)), and is frozen in the remaining states. The effect is that the SUT, from the point of view of the SUT based on operations synchronous with TCK, will see steps (iii), (vi), and (ix) as a continuous bitstream.

Further with respect to the first example, the "bypass sequence" is a property of the scan segment, and can be, for instance, a given sequence (all

zeros, all ones, or any other suitable sequence), or "don't care", where it is up to the TGT to decide such sequence.

As a second example, assume that the three segments of the system (A, B, and C) are implemented on different JTAG devices (in one or more cards). In this second example, once the three segments are defined in memory, the TISA operations would become:

i. Set Segment A states: StartState=RunTest/Idle, ScanState=ShiftIR, EndState=ShiftIR (gateTCK indicator);
ii. Set Segment A ScanLengthField to length of Segment A IR;
iii. Run ScanBlock with BYPASS instruction pattern for Segment A;
iv. Set Segment B states: StartState=ShiftIR, ScanState=ShiftIR, EndState=ShiftIR (gateTCK indicator);
v. Set Segment B ScanLengthField to length of Segment B IR;
vi. Run ScanBlock with EXTEST instruction pattern for Segment B;
vii. Set Segment C states: StartState=ShiftIR, ScanState=ShiftIR, EndState=RunTest/Idle;
viii. Set Segment C ScanLengthField to length of Segment C IR;
ix. Run ScanBlock with BYPASS instruction pattern for Segment C;
x. Set Segment A states: StartState=RunTest/Idle, ScanState=ShiftDR, EndState=ShiftDR (gateTCK);
xi. Set Segment A ScanLengthField to length of Segment A selected DR (1 bit BYPASS DR);
xii. Run ScanBlock with BYPASS data pattern for Segment A;
xiii. Set Segment B states: StartState=ShiftDR, ScanState=ShiftDR, EndState=ShiftDR (gateTCK);
xiv. Set Segment B ScanLengthField to length of Segment B selected DR (n bit BSR DR to pins);
xv. Run ScanBlock with EXTEST data pattern for Segment B;
xvi. Set Segment C states: StartState=ShiftDR, ScanState=ShiftDR, EndState=RunTest/Idle;

xvii. Set Segment C ScanLengthField to length of Segment C selected DR (1 bit BYPASS DR);

xviii. Run ScanBlock with BYPASS data pattern for Segment C.

In comparing the first example and the second example, it will be understood that the additional complexity associated with the second example comes from the need to use the Instruction Register (IR) of each JTAG device to select/deselect the segments. In that case, unused segments are directly taken out of the chain by putting the related JTAG device in the BYPASS mode of the 1149.1 standard (as indicated in steps (iii) and (xvii) of the second example).

It will be appreciated that all compositions of the above two examples are possible, with any number of segments defined on one or more JTAG devices. It will be further appreciated that the above-two examples are merely examples provided for the purpose of illustrating use of the Scan Segments Level for testing a system under test, and, thus, that embodiments in which the Scan Segments Level is used for testing a system under test are not intended to be limited by these examples.

In such embodiments, the actual sequence of TISA instructions can have multiple origins, including one or more of the following: (1) the TISA instructions may be statically computed by the TGT, in which case, each time the user wants to access a segment, the entire chain must be scanned (it will be appreciated that, while this solution is not optimized for scan time, it can be useful for embedded systems with limited computational resources and little or no time constraints); (2) the TISA instructions may be issued by a software scheduler, which receives access requests and composes them into scan operations; and/or (3) the TISA instructions may be issued by a hardware scheduler (e.g., such as, but not limited to, what is done for instruction reordering and bypass in some high-performance processors). It will be appreciated that TISA instructions associated with Scan Segments Level control may be issued in any other suitable way, which may include a

combination of the methods described above and/or one or more other suitable methods which may be used in place of or in addition to one or more of the methods described above.

The Scan Segments Level abstraction level is a powerful tool for handling dynamic topologies, such as the ones proposed by the IEEE P1687 standard and other dynamic topologies. If a section of the scan chain can be taken in and out the active scan path (e.g., using an SIB cell proposed by the IEEE P1687 standard or any other suitable hierarchy-enabling component(s)), that section can be marked as one (or more) segments. The testing scheduler then has knowledge, from the system state, as to whether or not this segment(s) is active, and, therefore, if the segment should be included in the TISA instruction scheduling. It will be appreciated by those skilled in the art and informed by the teachings herein that this principle also may be used for other dynamic elements, such as hot-swap boards (e.g., by detecting their presence from a status register) or any other suitable dynamic elements.

FIG. 18 depicts a high-level block diagram of one embodiment of a method for testing a portion of a system under test via a scan chain of the system under test using Scan Segments Level abstraction of the scan chain.

Although primarily depicted and described herein as being performed serially, at least a portion of the steps of method 1800 may be performed contemporaneously, or in a different order than depicted and described with respect to FIG. 18.

At step 1802, method 1800 begins.

At step 1804, the scan chain is decomposed into a plurality of segments. The scan chain is composed of a plurality of elements, and each segment includes at least one of the elements of the scan chain. The scan chain may be decomposed into segments in any suitable manner, as described hereinabove. As described herein, decomposition of the scan chain into segments may be applied anywhere in the development flow (e.g.,

by the test developer, by the test generation tool, by an embedded circuit model, and the like).

At step 1806, a set of instructions is generated. The set of instructions includes processor instructions associated with an ISA and test instructions for testing the portion of the system under test. The test instructions include, for each of the segments of the scan chain, at least one scan operation to be performed on the segment. The test instructions may be any type of test instructions, such as conventional test instructions, test instructions of a TISA, and the like, and, thus, may be generated in any suitable manner. The set of instructions may be generated in any suitable manner (e.g., in a manner the same as or similar to as depicted and described hereinabove respect to

At step 1808, the set of instructions is executed for testing the portion of the system under test. The set of instructions may be executed in any suitable manner, which may depend on the type of instructions of the set of instructions.

At step 1810, method 1800 ends.

Although primarily depicted and described herein with respect to embodiments in which embodiments of TISA are used to enable scan operations to be performed at the Scan Segments Level, it will be appreciated that one or more of the Scan Segments Level embodiments depicted and described herein also may be provided in environments using TISA-like instructions architectures, non-TISA instruction architectures and/or non-TISA testing environment implementations, and the like.

Although references are made herein to "the TISA" for purposes of describing the enhanced system testing capabilities enabled by exemplary embodiments of TISAs which may be formed and utilized as depicted and described herein, it will be appreciated that many different TISAs may be formed depending on various factors, such as one or more of the ISA of the processor for which the TISA is formed, characteristics of the SUT for which the TISA is formed, characteristics of the test algorithm the TISA is supposed

to execute, and the like, as well as various combinations thereof. Thus, references made herein to "the TISA" also may be read more generally as "a TISA" in that many different types of TISAs may be formed.

A position-based scheduling capability for supporting testing of Joint Test Action Group (JTAG) hardware is depicted and described herein.

In one embodiment, the position-based scheduling capability utilizes various features of TISA. In TISA, the scan chain is seen as a composition of sub-elements called segments, which results in the following advantages: (1) the operations on a segment are local (i.e. they are independent from the topology of which the segment is a part) and (2) an overall scan operation is composed by an ordered series of scan operations on the segments of which the scan chain is composed. It is noted that, in such an environment, the notion of "vector" is no longer needed; rather, only the position of the segment inside the scan chain is needed. As such, in one embodiment, the position-based scheduling capability is configured such that elements needing access to portions of the scan chain are configured to provide access requests to a scheduler which is configured to determine a schedule according to which the elements may access the requested portions of the scan chain, thereby obviating the need for each of the elements to compute the complete vector for the entire scan chain.

FIG. 19 depicts one embodiment of a scheduler execution architecture configured to provide position-based scheduling for testing a system under test.

As depicted in FIG. 19, scheduler execution architecture 1900 includes a system under test (SUT) 1910, a TISA processor 1920, a scheduler 1930, and a user application 1940.

The SUT 1910 includes a JTAG scan chain 1911 accessible via the TISA processor 1920 and a TAP which is omitted for purposes of clarity. The JTAG scan chain 1911 traverses four devices 1912. The four devices 1912 are denoted, in the direction from TDI to TDO, as Device 4, Device 3, Device

2, and Device 1, each of which includes a plurality of registers 1913. The SUT 1910 includes eight instruments 1915 disposed on the devices 1912 and accessible via the JTAG scan chain 1911. The eight instruments 1915 are denoted, in the direction from TDI to TDO, as Instruments 4.1 and 4.1 (which are disposed, in series, on Device 4), Instruments 3.2 and 3.1 (which are disposed, in series, on Device 3), Instrument 2 (which is disposed on Device 2), and Instruments 1 1.3, 1.2, and 1.1 (which are disposed, in series, on Device 1). The eight instruments 1915 are considered to be separate segments of the JTAG scan chain 1911 (e.g., as depicted and described herein with respect to one or more of FIGs. 1 through 18). The segments may be controlled individually and, thus, result in scan segment composition 1919 depicted in FIG. 19 (i.e., the JTAG scan chain 1911 may be considered to be composed of a sequence of eight scan segments which correspond to the eight instruments 1915, rather than being composed of each of the registers 1913 of each of the devices 1912 of the JTAG scan chain 1911).

The TISA processor 1920 is configured to receive TISA operations from user application 1940, based on a schedule determined by scheduler 1930, for testing of SUT 1910. The TISA operations may include TISA instructions (and, optionally, also may include other types of information). The TISA processor 1920 is configured to process the TISA operations to generate the associated input test data to be applied to SUT 1910 and, optionally, to receive output test data generated by SUT 1910. It is noted that, for at least some embodiments, the operation of TISA processor 1920 may be better understood by way of reference to FIGs. 1 - 18. It is noted that, when the operations are executed by the TISA processor 1920 and the input test data is updated, the TISA operations will be aligned within the correct positions within the JTAG scan chain 1911 (i.e., the TISA operations will be associated with the scan segments for which the TISA operations are intended, respectively). An example is depicted and described with respect to FIG. 20. Namely, FIG. 20 depicts an exemplary TISA segment concatenation

tor an exemplary scan chain composition. As described hereinabove, a TISA processor (such as TISA process 1920 depicted and described with respect to FIG. 19) is configured to execute operations on partial sections of the scan chain (the segments), and to freeze all signals, including the clock, between operations. In FIG. 20, lines TCK, TMS, TDI depict the JTAG outputs of a TISA processor, while TDI depict the input from a System Under Test. In FIG. 20, "FSM" depicts the state of the JTAG Finite State Machine belonging to the SUT, and it can be seen how its state transitions are independent from the pauses between segment operations. In FIG. 20, the last two lines show how the input/output streams coming from TDI and TDO, respectively, are distributed between the different segments.

The scheduler 1930 is configured to receive access requests from the user application 1940, perform position-based scheduling of the access requests, and respond to the user application 1940 with access responses indicative as to when the user application 1940 may perform TISA operations (associated with the access requests/responses, respectively) on portions of the JTAG scan chain 1911 of SUT 1910

The scheduler 1930 includes one or more scheduler core processes 1932 and a Circuit Model 1934.

The scheduler core processes 1932 are processes configured to perform position-based scheduling of the access requests received at the scheduler 1930 using the Circuit Model 1934. An exemplary scheduler core process 1932 is depicted and described with respect to FIG. 22. The scheduler core processes 1932 may be better understood by first considering the Circuit Model 1934.

The Circuit Model 1934 is a simplified model of the SUT 1910 that is specified in terms of the scan segments of which JTAG scan chain 1911 of SUT 1910 is composed.

The Circuit Model 1934 is configured to specify the order of the scan segments of the SUT 1910 (e.g., as depicted in scan segment composition

1919). The Circuit Model 1934 also may include any other information suitable for use by scheduler 1930 in determining scheduling of access requests/responses for execution of the corresponding TISA operations by TISA processor 1920.

In one embodiment, the Circuit Model 1934 is implemented using a tree structure. In this embodiment, each segment of which the JTAG scan chain 1911 is composed is represented as a leaf node in the tree, and the remainder of the tree includes super-segments (disposed over one or more hierarchical levels) which represent the hierarchical nature of the JTAG scan chain 1911. The root node of the tree represents the SUT 1910. It is noted that the Circuit Model 1934 as represented by the tree has a number of properties, including: (1) each node of the tree can be uniquely identified according to its path in the tree, (2) the Circuit Model 1934 is intrinsically hierarchal, such that systems may be composed by simply adding the entire sub-tree to the Circuit Model 1934, and (3) a simple depth-first (also known as post-order) traversal of the tree provides the correct order of segment accesses on the JTAG scan chain 1911 of SUT 1910. An exemplary scan chain composition and its associated Circuit Model for use as Circuit Model 1934 of SUT 1910 are depicted in FIG. 21.

FIG. 21 depicts an exemplary scan chain composition and associated Circuit Model for the system under test of FIG. 19.

As depicted in FIG. 21, exemplary scan chain composition 2110 is configured such that Devicel is composed of three segments corresponding to Instruments 1.1, 1.2, and 1.3, Device2 is composed of one segment corresponding to Instrument 2, Device 3 is composed of two segments corresponding to Instruments 3.1 and 3.2, and Device 4 is composed of two segments corresponding to Instruments 4.1 and 4.2.

As further depicted in FIG. 21, exemplary Circuit Model 2120 is a tree-based model of exemplary scan chain composition 2110. The root of exemplary Circuit Model 2120 (denoted as SUT) corresponds to SUT 1910.

The next level of exemplary Circuit Model 2120 include four nodes (denoted as Device 1, Instrument 2, Device 3, and Device 4, where the nodes for Devices 1, 3, and 4 each have additional nodes subtending therefrom and the node for Instrument 2 is a leaf node. The bottom level of exemplary Circuit Model 2120 includes seven leaf nodes corresponding to the seven segments of which some of the Instruments 1915 are composed (namely, the Device 1 node is a parent to three leaf nodes which represent Instruments 1.1, 1.2, and 1.3 of which Device 1 is composed; the Device 3 node is a parent to two leaf nodes which represent Instruments 3.1 and 3.2 of which Device 3 is composed; the Device 4 node is a parent to two leaf nodes which represent Instruments 4.1 and 4.2 of which Device 4 is composed). In the Circuit Model 2120, the scan segments are depicted as rectangular-shaped nodes and the remaining nodes are depicted as oval-shaped nodes.

In the Circuit Model 2120, each scan segment can be uniquely identified by its path in the tree. For example, the segment that represents Instrument 1.3 is uniquely identified by the path SUT.Devicel .Instrument .3., the segment that represents Instrument 2 is uniquely identified by the path SUT.Instrument2, the segment that represents Instrument 4.2 is uniquely identified by the path SUT.Device4.Instrument4.2, and so forth. As such, the exemplary scan chain composition 2110 may be obtained from exemplary Circuit Model 2120 by performing a depth-first traversal of the exemplary Circuit Model 2120. In one embodiment, in order to assist processing using such paths, the scheduler 1930 may be configured to associated unique identifiers with the paths such that the unique identifiers may be used to represent the paths, respectively (e.g., using unique integer identifiers which may be easier to handle than character strings, although it will be appreciated that any suitable type of unique identifiers may be used to represent the paths).

The scheduler 1930 receives unordered access requests from user application 1940, determines scheduling of the unordered access requests

based on the scan segment composition 1919 as determined by scheduler 1930 from Circuit Model 1934 of SUT 1910 which is available to scheduler 1930, and provides access responses to user application 1940 where the access responses are indicative as to when the user application 1940 may provide associated TISA operations (associated with the access requests and access responses, respectively) to TISA processor 1920 for execution. In this sense, scheduler 1930 processes an unordered set of access requests received from user application 1940 and generates an ordered set of access responses (e.g., ordered in accordance with scan segment composition 1919 as determined by scheduler 1930 from Circuit Model 1934 of SUT 1910), such that scheduler 1930 may then instruct user application 1940 as to the order in which TISA operations associated with the access requests may be provided to TISA processor 1920 for processing by TISA processor 1920.

The scheduler 1930 may be configured to perform position-based scheduling of access requests using the concept of critical section, which may be used to arbitrate mutual exclusive access in multi-tasking operating systems. An exemplary process by which scheduler 1930 may perform position-based scheduling of access requests using the concept of critical section is depicted and described with respect to FIG. 22.

The user application 1940 is configured to issue access requests associated with testing of SUT 1910. The user application 1940 may issue access requests for any of the instruments 1915. The user application 1940 does not have knowledge of the scan segment composition 1919 of SUT 1910; rather, the user application 1940 merely issues access requests without accounting for the order in which the TISA operations associated with the access requests may be or should be applied to the SUT 1910. The access requests issued by user application 1940 are received by scheduler 1930 for scheduling based on the scan segment composition 1919 as determined by the scheduler 1930 from the circuit model of the SUT 1910. The user application 1940 is further configured to receive access responses from the

scheduler 1930 in response to the access requests, and to provide TISA operations to the TISA processor 1920 in response to the access responses. An exemplary process by which the user application 1940 may request and receive access to portions of JTAG scan chain 1911 using the concept of critical section is depicted and described with respect to FIG. 22.

As depicted in FIG. 19, user application 1940 includes a plurality of user application elements 1941_1 - 1941_N (collectively, user application elements 1941), which may operate in a concurrent fashion. The access request sent by the user application 1940 may be sent by ones of the user application elements 1941 and, similarly, the access responses received and processed by the user application 1940 may be received by processed by ones of the user application elements 1941 from which the associated access requests are sent. The user application elements 1941 may be application threads or any other hardware and/or software based elements configured for use in sending access requests and receiving and processing access responses.

As depicted in FIG. 19, user application 1940 sends unordered access requests requesting access to segments of JTAG scan chain 1911. The scheduler 1930 receives the access requests from user application 1940. The set of unordered access requests includes access requests for the following segments in the following initial order: Instrument 3.2, Instrument 1.1, Instrument 3.1, Instrument 2, Instrument 4.2, Instrument 1.2, Instrument 4.1, and Instrument 1.3. The scheduler 1930 determines scheduling of the access requests using Circuit Model 1934. The scheduler 1930 determines the ordering of the scan segments of JTAG scan chain 1911 (i.e., scan segment composition 1919) using the Circuit Model 1934. The scheduler 1930 determines the ordering of the access requests based on the scan segment composition 1919 determined using Circuit Model 1934. The scheduler 1930 responds to the access requests with respective access responses provided to the user application. The set of ordered access

responses includes access responses for the following segments in the following order: Instrument 1.1, Instrument 1,2, Instrument 1.3, Instrument 2, Instrument 3.1, Instrument 3.2, Instrument 4.1, and Instrument 4.2. The user application 1940 receives the set of ordered access responses from scheduler 1930. The user application 1940 then performs TISA operations on the JTAG scan chain 1911 of SUT 1910 based on the ordering of the segments as specified by the set of ordered access responses (namely, in the following order: Instrument 1.1, Instrument 1,2, Instrument 1.3, Instrument 2, Instrument 3.1, Instrument 3.2, Instrument 4.1, and Instrument 4.2).

FIG. 22 depicts one embodiment of a process for performing position-based scheduling of access requests using the concept of critical section.

At step 2210, the user application 1940 sends an access request (depicted as REQUEST) to the scheduler 1930.

The user application 1940 may initiate the access request whenever the user application needs to access a resource of the SUT 1910.

The access request is configured to request access by the user application 1940 to a resource of the SUT 1910. The resource of the access request may be any suitable type of resource (e.g., a scan segment of the SUT 1910 and/or any other suitable type of resource).

The access request may identify the resource of SUT 1910 in any suitable manner. In one embodiment, for example, the access request may include a unique identifier (which may be denoted herein as a Unique Identifier (UID)) specifying the resource of the SUT 1910 to which the user application 1940 is requesting access.

It is noted that the access request does not need to include the associated test data to be scanned for the resource to which access is requested, at least because the scheduler 1930 is configured to return an associated access response to the user application 1940 (at which time user application 1940 may then use the relevant test data, available at and/or to

the user application 1940, when providing the associated TISA operation(s) to TISA processor 1920).

The user application 1940 then waits for an access response from the scheduler 1930, during which time the user application 1940 may operate in any suitable manner (e.g., entering a sleep state or other suitable waiting state, initiating one or more additional access requests to scheduler 1930, performing one or more other processing functions, and the like).

At step 2220, the scheduler 1930 determines authorization of the access request.

The scheduler 1930 is configured to receive multiple unordered access requests for access to resources of SUT 1910 (including the REQUEST depicted in FIG. 22 and one or more other access requests omitted from FIG. 22 for purposes of clarity) and process the multiple unordered access requests for determining scheduling of the unordered access requests based on the scan segment composition 1919 as determined by scheduler 1930 from Circuit Model 1934 of SUT 1910. In this sense, for the specific access request of FIG. 22, the scheduler 1930 is configured to determine the time at which the user application 1940 is granted access to the resource of the SUT 1910 that was specified in the access request of FIG. 22 (within the context of authorizing other access requests for resources of SUT 1910 that are received from user application 1940 which, again, are omitted from FIG. 22 for purposes of clarity).

The scheduler 1930 processes the Circuit Model 1934 to determine scheduling of the access request. It is noted that scheduler 1930 may explore the Circuit Model 1934 in any suitable manner. For example, exploration of the Circuit Model 1934 may be launched following a precise strategy (e.g., all leaves are marked as pending, time-triggered, explicit requests, and the like). For example, upon receipt of the access request for the scan segment, the scheduler 1930 marks the associated leaf node of Circuit Model 1934 (which corresponds to the requested scan segment) as pending. For example, if

during exploration a segment is encountered that is not pending, a default sequence is shifted instead (e.g., the sequence can be fixed, can be included inside of the leaf (e.g., defined by the instrument provider), and the like). It is noted that, in the case when an instrument operation might be perturbed by scan operations, the leaf node may ask the scheduler 1930 to inhibit accesses for a certain length of time.

At step 2230, the scheduler 1930 sends an access response to the user application 1940 (depicted as RESPONSE), responsive to the access request (REQUEST) received from the user application 1940.

The access response includes the unique identifier provided in the access request (again, which may be denoted herein as a UID), which specifies the resource(s) of the SUT 1910 to which the user application 1940 has been granted access by scheduler 1930.

The access response indicates to the user application 1940 that it is the only entity with access rights to the resource of SUT 1910 that is identified by the UID (i.e., the resource of SUT 1910 that is identified by the UID is locked for use by the user application 1940 to which access was granted by the scheduler 1930). Accordingly, in conjunction with sending the access response, the scheduler 1930 locks the resource of SUT 1910 that is identified by the UID. The scheduler 1930 may lock the resource of SUT 1910 that is identified by the UID in any suitable manner (e.g., via changing of one or more values associated with the node of the Circuit Model 1934 that is associated with the resource of SUT 1910 that is identified by the UID, or in any other suitable manner).

At step 2240, after receiving the access response, the user application 1940 has access rights to the resource of SUT 1910 that is identified by the UID (with a guarantee that it is the only entity that currently has access rights to the resource of SUT 1910 that is identified by the UID) and performs one or more TISA operations on the resource of SUT 1910 that is identified by the UID. The operations may include sending one or more TISA operations,

related to the resource of SUT 1910 that is identified by the UID, from user application 1940 to TISA processor 1920 for execution by TISA processor 1920. The operation(s) performed by user application 1940 may include any suitable number of TISA accesses via TISA processor 1920 to the SUT 1910. The user application 1940 may continue to use the resource of SUT 1910 that is identified by the UID until it no longer needs access to the resource of SUT 1910 that is identified by the UID, at which time the user application 1940 may initiate release of the resource of SUT 1910 that is identified by the UID.

At step 2250, the user application 1940 sends a resource release message (depicted as RELEASE) to scheduler 1930. The user application 1940 sends the resource release message when the user application no longer needs access to the resource of SUT 1910 for which access was originally requested. The resource release message includes the unique identifier provided in the access request (again, which may be denoted herein as a UID), which specifies the resource of the SUT 1910 to which the user application 1940 has been granted access by scheduler 1930. The scheduler 1930, upon receiving the resource release message, releases the resource of SUT 1910 that is identified by the UID. The scheduler 1930 may release the resource of SUT 1910 that is identified by the UID in any suitable manner (e.g., via changing of one or more values associated with the node of the Circuit Model 1934 that is associated with the resource of SUT 1910 that is identified by the UID).

It is noted that, although primarily depicted and described with respect to an embodiment in which an access request is configured to request access to a single resource, access to multiple resources may be requested via one or more associated access requests.

It is noted that process 2200 is repeated for each access request that is initiated by user application 1940, thereby enabling the scheduler 1930 to authorize multiple access requests in a manner that is coherent with JTAG scan chain 1911 and, similarly, enabling user application 1940 to access

portions ot JTAG scan chain 1911 in an order that is based on scan segment composition 1919 of JTAG scan chain 1911 without requiring user application to computer the complete vector for the entire JTAG scan chain 1911. In this sense, it is noted that from the perspective of user application 1940, process 2200 can be performed concurrently by various one of the user application elements 1941 as various ones of the user application elements 1941 issue access requests and receive and process associated access responses. It is noted that process 2200 may be implemented in any suitable manner. In one embodiment, for example, process 2200 may be implemented using semaphores and mutexes. For example, in reference to common use of computer science techniques and, more specifically, to Dijkstra's semaphore formulation, a REQUEST 2210 is equivalent to a "P" operation while a RELEASE 2250 is equivalent to a "V" operation.

It is noted that various embodiments of the position-based scheduling capability enable simplification of the user application 1940. In the absence of the position-based scheduling capability, the user application 1940 would need to maintain an associated Circuit Model that models the entire SUT 1910. For example, the user application 1940 would need to maintain a Circuit Model which provides a description of SUT 1910 that includes (1) for each device in the SUT 1910, a structure description of the device (e.g., the register(s) in the device, the instruction(s) used to access the register(s), the length(s) of the register(s), and the like) and (2) a linked list which describes the connections between the devices of SUT 1910. Similarly, in the absence of the position-based scheduling capability, when the user application 1940 wants to write something to a particular register in SUT 1910 the user application 1940 would need to find the device, find the register in that device, and update the register, and then, for each associated scan operation, also would need to proceed through the linked list of devices in order to gather all of the vector information for the current register that is active, write out a super-vector representing all of the devices in the scan chain of SUT 1910,

read the output data, process the output data in order to put the output data back into the Circuit Model of SUT 1910 in order to analyze the output data. The position-based scheduling capability obviates the need for user application 1940 to maintain such a Circuit Model that describes the entire SUT 1910; rather, the user application 1940 would merely need to be aware of portions of SUT 1910 in which it is interested (e.g., as denoted by its UID). The position-based scheduling capability also obviates the need for user application 1940 to deal with the entire scan chain of the SUT 1910 when performing operations on SUT 1910 (e.g., obviates the need for the user application 1940 to construct a super input vector for the entire scan chain of SUT 1910 and process a super output vector for the entire scan chain of SUT 1910 when performing operations on SUT 1910, including when performing operations on only a portion of SUT 1910). Rather, in TISA, the use of segments (and the associated representation of the scan segments in a Circuit Model that is a tree-based representation in which a depth-first traversal of the tree yields the composition of the scan segments of which the scan chain of SUT 1910 is composed), enables delegation of operations from user application 1940 in the TISA model (e.g., where an operation intended for a particular device or register represented by a particular scan segment can be handled by sending the operation) to the node of the Circuit Model which represents that scan segment such that the node can perform a scan operation with TISA that is specific only to that device or register. In other words, it is not necessary to construct the entire vector for the entire scan chain of SUT 1910 in order to perform the intended operation. It is noted that this is a huge boost to portability for the user application 1940, which does not need to be adapted to the actual scan chain. For example, if the user application 1940 is only interested in interacting with Device 4 of SUT 1910 (e.g., user application 1940 is an emulation application and Device 4 is a microprocessor chip), user application 1940 is (1) not required to be aware of Device 3, Device 2, or Device 1 of SUT 1910 and, thus, is not required to

maintain a Circuit Model including information for Device 3, Device 2, or Device 1 of SUT 1910 and (2) not required to construct a super input vector for the entire scan chain of SUT 1910 and process a super output vector for the entire scan chain of SUT 1910 when performing operations on SUT 1910. Rather, the user application 1940 merely needs to send an access request to scheduler 1930 where the access request indicates the operation that the user application 1940 would like to perform on Device 4 (e.g., update a particular register in Device 4, or any other suitable operation). In this manner, handling of requested operations is transformed from a modelling problem (including the associated processing and efficiency problems that are associated therewith when trying to construct a super input vector from structure description and linked list information and process an associated super output vector in order to analyze the desired results) into a scheduling problem (which may be handled by scheduler 1930 using various computer science functions). Therefore, various embodiments of the position-based scheduling capability are more efficient than existing testing capabilities.

Although omitted for purposes of clarity, it is noted that the various elements of scheduler execution architecture 1900 may be implemented using one or more devices and, similarly, that communication between the various elements of scheduler execution architecture 1900 may be performed in any suitable manner (e.g., depending on the number of devices used, the locations of the device, and the like). It is further noted that various elements of scheduler execution architecture may be implemented using hardware and/or software. For example, scheduler 1930 and user application 1940 each may be implemented in hardware or as a software-based module stored in one or more memories and configured for execution by one or more processors (e.g., TISA processor 1920 and/or any other suitable processor(s), which may depend on the locations of TISA processor 1930, scheduler 1930, and user application 1940).

In one embodiment, for example, TISA processor 1920, scheduler 1930, and user application 1940 may be implemented in separate devices which may communicate with each other via one or more communication networks (e.g., via an Ethernet network, the Internet, and the like, as well as various combinations thereof).

In one embodiment, for example, TISA processor 1920 and scheduler 1930 may be co-located in a single device and user application 1940 may be implemented in a separate device. For example, scheduler 1930 may be a software-based module stored in one or more memories accessible to one or more processors (e.g., TISA processor 1920 and/or one or more other processors which may cooperate with TISA processor 1920) configured to execute the scheduler 1930 in order to provide the various functions depicted and described herein. For example, user application 1940 may be implemented in hardware or as a software-based module stored in one or more memories and configured for execution by one or more processors to provide the various functions depicted and described herein. For example, communication between scheduler 1930 and user application 1940 may be via one or more communication networks (e.g., via an Ethernet network, the Internet, and the like, as well as various combinations thereof).

In one embodiment, for example, TISA processor 1920 and user application 1940 may be co-located in a single device and scheduler 1930 may be implemented in a separate device. For example, user application 1940 may be a software-based module stored in one or more memories accessible to one or more processors (e.g., TISA processor 1920 and/or one or more other processors which may cooperate with TISA processor 1920) configured to execute the user application 1940 in order to to provide the various functions depicted and described herein. For example, scheduler 1930 may be implemented in hardware or as a software-based module stored in one or more memories and configured for being executed by one or more processors to provide the various functions depicted and described herein.

For example, communication between user application 1940 and scheduler 1930 may be via one or more communication networks (e.g., via an Ethernet network, the Internet, and the like, as well as various combinations thereof).

In one embodiment, for example, scheduler 1930 and user application 1940 may be co-located in a single device and TISA processor 1920 may be implemented as a separate device. For example, scheduler 1930 and user application 1940 each may be implemented in hardware or as a software-based module stored in one or more memories and configured to be executed by one or more processors to provide the various functions depicted and described herein. For example, communication between user application 1940 and TISA processor 1920 may be via one or more communication networks (e.g., via an Ethernet network, the Internet, and the like, as well as various combinations thereof).

In one embodiment, for example, TISA processor 1920, scheduler 1930, and user application 1940 may be co-located in a single device. For example, scheduler 1930, and user application 1940 may be software-based modules stored in one or more memories accessible to one or more processors (e.g., TISA processor 1920 and/or one or more other processors which may cooperate with TISA processor 1920) configured to execute the scheduler 1930 and user application 1940 to provide the various functions depicted and described herein. An exemplary embodiment is depicted and described with respect to FIG. 23.

FIG. 23 depicts one embodiment of a scheduler execution architecture configured to provide position-based scheduling for testing a system under test.

As depicted in FIG. 23, scheduler execution architecture 2300, which is similar to the scheduler execution architecture 1900 of FIG. 19, includes a system under test (SUT) 2310, a main CPU 2320, a scheduler 2330, and a user application 2340. For purposes of clarity, it is noted that SUT 2310 is equivalent to SUT 1910, main CPU 2320 is equivalent to TISA processor

1920, scheduler 2330 is equivalent to scheduler 1930, and user application 2340 is equivalent to user application 1940 (where segment threads 2341_1 to 2341_N are equivalent to user application elements 1941_1 to 1941_N, respectively).

The SUT 2310 includes a JTAG scan chain 2311 accessible via the main CPU 2320 and a TAP (which is omitted for purposes of clarity). The JTAG scan chain 2311 has an associated scan chain composition 2318 composed of a plurality of scan segments 2319_1 - 2319_M (collectively, scan segments 2319).

The main CPU 2320 supports a TISA 2322, such that the main CPU 2320 is configured to function as a TISA processor (e.g., in a manner similar to TISA processor 1920 depicted and described with respect to FIG. 19).

The main CPU 2320 has access to a memory 2324, which stores scheduler 2330 and user application 2340.

The scheduler 2330 is configured to receive access requests from the user application 2340, perform position-based scheduling of the access requests, and respond to the user application 2340 with access responses indicative as to when the user application 2340 may perform TISA operations on portions of the JTAG scan chain 2311 of SUT 2310 that are associated with the access requests/responses, respectively. For example, the TISA operations may include providing TISA operations to the TISA processor 2320 for processing.

The scheduler 2330 includes one or more scheduler core processes 2332 and a Circuit Model 2334.

The scheduler core processes 2332 are processes configured to perform position-based scheduling of the access requests received at the scheduler 2330 using the Circuit Model 2334. The scheduler 2330 also includes an access Application Programming Interface (API) 2333 which provides an API via which user application 2340 accesses scheduler core processes 2332. The access API 2333 is configured for enabling the user

application 2340 to communicate with the scheduler 2330 using an access scheduling protocol (e.g., access scheduling protocol 2200 depicted and described with respect to FIG. 22). The scheduler core processes 2332 may be configured to execute portions of the access scheduling protocol depicted and described as being performed by scheduler 1930.

The Circuit Model 2334 is a simplified model of the SUT 2310 that is specified in terms of the scan segments 2319 of which the JTAG scan chain 2311 of SUT 2310 is composed. The scheduler 2330 also includes a circuit model Application Programming Interface (API) 2335 which provides an API via which user application 2340 accesses Circuit Model 2334. The circuit model API 2335 is configured for enabling the user application 2340 to build and modify the Circuit Model 1934. It is noted that the Circuit Model 1934 may be built/modified via circuit model API 2335 as depicted and described with respect to exemplary circuit model 2110 of FIG. 21.

The user application 2340 is configured to issue access requests associated with testing of SUT 2310. The user application 2340 may issue access requests for any of the scan segments 2319 of SUT 2310. The user application 2340 does not have knowledge of the scan segment composition 2318 of SUT 2310; rather, the user application 2340 merely issues access requests without accounting for the order in which the access requests may be or should be applied to SUT 2310. The access requests issued by user application 2340 are received by scheduler 2330 for scheduling based on the scan segment composition 2318 as determined by the scheduler 2330 from the circuit model of the SUT 2310.

The user application 2340 includes a plurality of scan segment threads 2342i - 2342_N (collectively, scan segment threads 2342) and a main thread 2344. The scan segment threads 2342 are supported by main thread 2344.

The scan segment threads 2342 of user application 2340 each are configured to communicate with scheduler core processes 2332 via access API 2333 using respective instances of the access scheduling protocol (e.g.,

the access scheduling protocol 2200 depicted and described with respect to FIG. 22). In this sense, the scan segment threads 2342 support multi-tasking, as one or more of the scan segment threads 2342 may be involved in access arbitration with scheduler 2330 via access API 2333 at any given time.

The main thread 2344 of user application 2340 is configured to build and modify Circuit Model 2334 of scheduler 2330 via circuit model API 2335 of scheduler 2330.

It is noted that the operation of user application 2340 and scheduler 2330 in testing the SUT 2310 via main CPU 2320 may be better understood when the scheduler execution architecture 2300 of FIG. 23 is reviewed in conjunction with the scheduler execution architecture 1900 of FIG. 19.

As noted herein, main CPU 2320 has access to memory 2324, which stores scheduler 2330 and user application 2340. In this sense, scheduler core processes 2332 of scheduler 2330 may be accessed and executed by main CPU 2320 for providing various functions of scheduler 2320 depicted and described herein. Similarly, segment threads 2342 of user application 2340 may communicate with main CPU 2320 for purposes of performing TISA accesses to JTAG scan chain 2311 of SUT 2300 via TISA 2322. Although primarily depicted and described with respect to use of a single memory 2324, it is noted that scheduler 1930 and/or user application 2340 may be provided using any suitable number of memories.

It is noted that, in at least some embodiments, scheduler execution architecture 2300 is compatible with various multi-tasking operating systems (OSs) and/or other types of frameworks. In one embodiment, for example, implementation of scheduler execution architecture 2300 using POSIX allows scheduler execution architecture 2300 to be run as a standalone application that incorporates a reduced thread scheduler. Although primarily depicted and described herein with respect to use of position-based scheduling capability for scheduling of static paths, it is noted that various embodiments of the position-based scheduling capability may support scheduling of

dynamic paths. In general, dynamic paths are one of the major limitations of traditional retargeting, primarily due to the fact that dynamic paths force heavy vector regeneration and/or modification.

In one embodiment of the position-based scheduling capability, a dynamic scan path of a SUT may be supported by enabling one or more dynamic segments to be inserted into and removed from the scan chain of the SUT as needed or desired. In one embodiment, a dynamic segment may be configured to be dynamically inserted into the scan chain or removed from the scan chain via a control entity defined for an SUT, wherein the dynamic segment is represented in the tree as a leaf node having an associated parent node representing the control entity. The control entity may, but is not required to, reside directly adjacent to the dynamic segment in the hierarchy. For example, the dynamic segment could be controlled by some other data register that is multiple segments away. For example, the dynamic segment could be controlled by a separate data register that is accessed via its own instruction in the 1149.1 TAP Controller that is no longer accessible in the scan chain of the SUT, but its state is preserved as defined by the 1149.1 standard. In other words, many different types of representations for control are possible and, thus, the control entity may be implemented in a number of different ways. It is noted that control entity is defined for the SUT in the sense that there exists a predefined description of control aspects of the control entity that can be modelled and controlled by the tool.

In one embodiment of the position-based scheduling capability, scheduling for a dynamic scan path may be supported by representing the dynamic path as a super-segment. This embodiment may be better understood by considering FIG. 24.

FIG. 24 depicts an exemplary scan chain composition and associated Circuit Model where the scan chain composition includes a dynamic segment.

As depicted in FIG. 24, an exemplary SUT 2410 supports a dynamic scan path in which an element can be inserted into and removed from a JTAG

scan chain 241 1 of the SUT 241 0 dynamically. The SUT 241 0 supports two different configurations depending on the value of a cell (illustratively, cell X) that is included in the JTAG scan chain 241 1 of SUT 241 0. In a first configuration of SUT 241 0 (depicted as SUT 241 0_1), the value of cell X is set such that only the cell X and a static section (denoted as static) are active within the JTAG scan chain 241 1. In a second configuration of SUT 241 0 (depicted as SUT 241 0_2), the value of cell X is set such that a dynamic section (denoted as dynamic) is added to the JTAG scan chain 241 1 between the cell X and the active section, i.e., the cell X, the dynamic section, and the static section (denoted as static) are active within the JTAG scan chain 241 1. The insertion of the dynamic section to JTAG scan chain 241 1 and removal of the dynamic section from JTAG scan chain 241 1 may be controlled by changing the value of cell X. It is noted that cell X may be implemented using any suitable element (e.g., using the Segment Insertion Bit (SIB) of the IEEE 1687 draft standard, or using any other suitable element(s)).

As further depicted in FIG. 24, exemplary SUT 241 0, which supports a dynamic JTAG scan path 241 1, may be represented using a Circuit Model 2420. In the Circuit Model 2420, cell X is represented as a super-segment 2421 (represented as the trapezium in FIG. 24). The cell X has a status associated therewith, such that super-segment 2421 has a status associated therewith. When a depth-first traversal of Circuit Model 2420 is performed, recursion through cell X is dependent on the status of cell X (e.g., recursion through cell X is allowed when the status of cell X is "open"; otherwise, cell X is treated as a leaf segment when the status of cell X is "closed").

In one embodiment, the associated scheduler (e.g., scheduler 1930 or scheduler 2330) is able to adapt on-the-fly based on the current status of Circuit Model 2420, without any need of retargeting.

In one embodiment, the associated scheduler is configured to detect a REQUEST to the "dynamic" node of the Circuit Model 2420 when the "dynamic" node of the Circuit Model 2420 is not active (when the status of cell

X is "closed" such that super-segment 2421 of cell X is closed) and, in response, to generate one or more scan segment operations to open the super-segment 2421 of cell X with which the non-active "dynamic" leaf node is associated (thereby causing the "dynamic" node of the Circuit Model 2420 to become active). More generally, in one embodiment, the scheduler is configured to detect a REQUEST to a non-active leaf node and, in response, to generate one or more scan segment operations to change the status of the associated super-segment (i.e., the super-segment with which the non-active leaf node is associated) from "closed" to "open" such that the non-active leaf node becomes active. It is noted that the operation of changing of state of a super-segment can be as simple or complex as needed, and in an object-oriented paradigm can be easily delegated by the scheduler 2330 to an implementation method (e.g., using the visitor design pattern, or any other suitable mechanism).

It is noted that, in such embodiments, any suitable set of status values may be used to control depth-first traversal of Circuit Model 2420. In one embodiment, for example, status values of "open" (indicative that traversal of the node is allowed), "closed" (indicative that recursion ends at this node and the state remains the same), and "pending" (indicative that recursion ends at this node and the state of the node is changed to "open" for the next iteration) may be supported. It is noted that different sets of status values also may be used.

It is noted that, in such embodiments, status of nodes within Circuit Model 2420 may be managed in any suitable manner. In one embodiment, for example, graph coloring may be used to manage the status of nodes of Circuit Model 2420, whereby each node of the Circuit Model 2420 is assigned a color which may be used by the depth-first traversal process to determine the action(s) to take with respect to the node.

Although primarily depicted and described with respect to use of the super-segment 2421 to regulate access to a single dynamic segment, the

super-segment 2421 may be used to regulate access to multiple dynamic segments. In this case, the state of the super-segment (i.e., the value of cell X) may determine which one or more dynamic segments are active and the order in which they are to be traversed.

FIG. 25 depicts one embodiment of a method for supporting position-based scheduling for testing a system under test.

As depicted in FIG. 25, steps are performed by the user application, scheduler, and TISA processor used for testing a system under test (e.g., user application 1940, scheduler 1930, and TISA processor 1920 depicted and described with respect to FIG. 19). Although primarily depicted and described as being performed serially in a particular order, at least a portion of the steps of method 2500 may be performed contemporaneously and/or in a different order than depicted in FIG. 25.

At step 2505, method 2500 begins. At step 2510, the user application 1940 generates unordered access requests requesting access to respective portions of the system under test. At step 2515, the user application 1940 sends unordered access requests to scheduler 1930. At step 2520, the scheduler 1930 receives the unordered access requests from user application 1940. At step 2525, scheduler 1930 determines scheduling of the unordered access requests using an associated circuit model. At step 2530, scheduler 1930 generates ordered access responses associated with the unordered access requests. At step 2535, the scheduler 1930 sends the ordered access responses to the user application 1940. At step 2540, the user application 1940 receives the ordered access responses from the scheduler 1930. At step 2545, the user application 1940 generates TISA operations associated with the ordered access responses. At step 2550, the user application 1940 sends the TISA operations to the TISA processor 1920. At step 2555, the TISA processor 1920 receives the TISA operations. At step 2560, the TISA processor 1920 processes the TISA operations. At step 2565, method 2500 ends. It is noted that the various steps of method 2500 may be better

understood when considered in conjunction with the description of FIGs. 19 - 24.

Although FIGs. 19 - 25 are primarily depicted and described herein with respect to supporting position-based scheduling in the testing input direction from the user application 1940 toward the SUT 1910 (e.g., for initiating tests on SUT 1910), it will be appreciated that the various functions of position-based scheduling may be performed in the opposite order to support position-based scheduling in the testing output direction from the SUT 1910 toward the user application 1940 (e.g., for receiving and analyzing the results of tests initiated for SUT 1910). In one embodiment, for example, processing may proceed as follows: (1) TISA processor 1920 receives testing output data from SUT 1910 via TDO, processes the testing output data to produce testing output results associated with TISA operations processed by TISA processor 1920 to produce the testing input data provided to SUT 1910, and propagates the testing output results toward user application 1940, and (2) user application 1940 receives the testing output results from TISA processor 1920 and processes the testing output results (e.g., for analyzing the results of tests initiated for SUT 1910). Although primarily depicted and described within the context of embodiments in which the user application is responsible for providing TISA operations to the TISA processor, it is noted that in one embodiment the scheduler may be responsible for providing TISA operations to the TISA processor. An exemplary embodiment is depicted and described with respect to FIG. 26.

FIG. 26 depicts one embodiment of a scheduler execution architecture configured to provide position-based scheduling for testing a system under test.

As depicted in FIG. 26, the scheduler execution architecture 2600 of FIG. 26 is similar to the scheduler execution architecture 1900 of FIG. 19 with the exception that the scheduler 1930, rather than the user application 1940, is configured to provide the TISA operations to the TISA processor 1920.

The user application 1940 sends unordered access requests to the scheduler 1930. Additionally, user application 1940 also provides associated test data for the access requests to scheduler 1930. The test data may be provided with the access requests and/or separate from the access requests. The access requests and associated test data may be provided from user application 1940 to scheduler 1930 in any suitable format (e.g., as packets, messages, and the like, as well as various combinations thereof).

The scheduler 1930 receives the unordered access requests and the associated test data of the access requests from user application 1940. The scheduler 1930, rather than providing ordered access responses to the user application 1940 (as depicted and described with respect to FIG. 19), determines scheduling of the access requests and issues appropriate TISA operations to TISA processor 1920 based on the determined scheduling of the access requests. The scheduler 1930 is configured to use the test data of the access requests in order to generate the TISA operations for the access requests, respectively. The scheduler 1930 issues the TISA operations to TISA processor 1920 based on the scheduling of the access requests / TISA operations as determined by scheduler 1930.

The TISA processor 1920 receives the TISA operations from scheduler 1930, and processing proceeds as depicted and described with respect to scheduler execution architecture 1900 of FIG. 19).

FIG. 27 depicts one embodiment of a method for supporting position-based scheduling for testing a system under test. As depicted in FIG. 27, steps are performed by the user application, scheduler, and TISA processor used for testing a system under test (e.g., user application 1940, scheduler 1930, and TISA processor 1920 depicted and described with respect to FIG. 19). Although primarily depicted and described as being performed serially in a particular order, at least a portion of the steps of method 2700 may be performed contemporaneously and/or in a different order than depicted in FIG. 27.

At step 2705, method 2700 begins. At step 2710, the user application 1940 generates unordered access requests requesting access to respective portions of the system under test. At step 2715, the user application 1940 sends unordered access requests and associated input test data for the access requests to scheduler 1930. At step 2720, the scheduler 1930 receives the unordered access requests from user application 1940. At step 2725, scheduler 1930 determines scheduling of the unordered access requests using an associated circuit model. At step 2730, scheduler 1930 generates TISA operations associated with the scheduled access requests. At step 2735, the scheduler 1930 sends the TISA operations to the TISA processor 1920. At step 2740, the TISA processor 1920 receives the TISA operations. At step 2745, the TISA processor 1920 processes the TISA operations. At step 2750, method 2700 ends. It is noted that the various steps of method 2700 may be better understood when considered in conjunction with the description of FIGs. 19 - 26.

Although FIGs. 26 and 27 are primarily depicted and described herein with respect to embodiments in which input testing data is provided from the user application 1940 to scheduler 1930 and scheduler 1930 generates the associated TISA operations to be provided to TISA processor 1920, it is noted that in at least some embodiments (1) the user application 1940 is configured to generate the TISA operations and to provide the TISA operations to the scheduler 1930 for scheduling of the TISA operations and (2) scheduler 1930 is configured to schedule the TISA operations received from user application 1940 and to provide the TISA operations to TISA processor 1920 based on the determined scheduling of the TISA operations. As with the embodiments depicted and described in FIGs. 26 and 27, the user application 1940 may be configured to send access requests to scheduler 1930 where, rather than just providing the input testing data for the access requests, the user application 1940 provides the TISA operations (and associated input testing data) for the access requests (e.g., as part of the access requests and/or separate from

the access requests, similar to the embodiments depicted and described with respect to FIGs. 26 and 27). It is noted that such embodiments obviate the need for the scheduler 1930 to generate the TISA operations, but at the expense of requiring a larger quantity of information to be sent from the user application 1940 to scheduler 1930.

Although FIGs. 26 and 27 are primarily depicted and described herein with respect to supporting position-based scheduling in the testing input direction from the user application 1940 toward the SUT 1910 (e.g., for initiating tests on SUT 1910), it will be appreciated that the various functions of position-based scheduling may be performed in the opposite order to support position-based scheduling in the testing output direction from the SUT 1910 toward the user application 1940 (e.g., for receiving and analyzing the results of tests initiated for SUT 1910). In one embodiment, for example, processing may proceed as follows: (1) TISA processor 1920 receives testing output data from SUT 1910 via TDO, processes the testing output data to produce testing output results associated with TISA operations processed by TISA processor 1920 to produce the testing input data provided to SUT 1910, and propagates the testing output results toward scheduler 1930, (2) scheduler 1930 receives the testing output results from TISA processor 1920 and propagates the testing output results toward user application 1940, and (3) user application 1940 receives the testing output results from scheduler 1930 and processes the testing output results (e.g., for analyzing the results of tests initiated for SUT 1910).

Although omitted for purposes of clarity, it is noted that communication between the various elements of scheduler execution architecture 2600 may be performed in any suitable manner. Many such embodiments are depicted and described with respect to FIG. 19. It is noted that, in the case of the scheduler execution architecture 2600, where the TISA processor 1920 and the scheduler 1930 are implemented as separate devices communication between the TISA processor 1920 and the scheduler 1930 may be via one or

more communication networks (e.g., via an Ethernet network, the Internet, and the like, as well as various combinations thereof).

Although primarily depicted and described herein with respect to embodiments in which the scheduler 1930 is implemented separate from the TISA processor 1920 and the TISA processor 1920 is implemented separate from the SUT 1910, it is noted that the scheduler execution architecture may be implemented in various other ways. In one embodiment, for example, the scheduler 1930 may be embedded on the TISA processor 1920. In one embodiment, for example, the TISA processor 1920 and the scheduler 1930 may be embedded within the SUT 1910. Various other arrangements are contemplated.

Although primarily depicted and described herein with respect to embodiments in which test operations are issued when the scheduler allows the test operations to be issued, in one embodiment issuance of test operations may be deferred using a deferred scheduling capability.

FIG. 28 depicts one embodiment of a deferred scheduler execution architecture configured to provide position-based scheduling for testing a system under test.

As depicted in FIG. 28, deferred scheduler execution architecture 2800 is a modified version of the scheduler execution architecture 1900 depicted and described with respect to FIG. 19.

The deferred scheduler execution architecture 2800 includes SUT 1910, scheduler 1930, and user application 1940, each of which may be configured to function as depicted and described with respect to the embodiments the scheduler execution architecture 1900 of FIG. 19. The deferred scheduler execution architecture 2800 also includes a deferred scheduling module 2801. The deferred scheduling module 2801 includes a TISA module 2810, a reordering buffer module 2820, a vector transformation module 2830, and a JTAG TAP 2840. The TISA module 2810 is disposed between user application 1940 and reordering buffer module 2820. The

reordering buffer module 2820 is disposed between TISA module 2810 and vector transformation module 2830. The vector transformation module 2830 is disposed between the reordering buffer module 2820 and the JTAG TAP 2840. The JTAG TAP 2840 is disposed between vector transformation module 2830 and SUT 1910 (or, optionally, one or more additional modules which may be configured to perform other types of operations).

It is noted that deferred scheduling module 2801 and, more specifically, the TISA module 2810, the reordering buffer module 2820, and the vector transformation module 2830, may be implemented using hardware and/or software.

In one embodiment, for example, TISA module 2810, reordering buffer module 2820, and vector transformation module 2830 may be implemented in hardware (e.g., using one or more hardware components).

In one embodiment, for example, TISA module 2810 and reordering buffer module 2820 are implemented in software (e.g., as one or more software modules configured to run on one or more processors or other suitable hardware) and vector transformation module 2830 is implemented in hardware (e.g., using one or more hardware components).

In one embodiment, for example, TISA module 2810 is implemented in software (e.g., as one or more software modules configured to run on one or more processors or other suitable hardware), and reordering buffer module 2820 and vector transformation module 2830 are implemented in hardware (e.g., using one or more hardware components).

In one embodiment, for example, TISA module 2810, reordering buffer module 2820, and vector transformation module 2830 are all implemented in software (e.g., as one or more software modules configured to run on one or more processors or other suitable hardware).

In at least some implementations, as indicated above, TISA module 2810 may be implemented as a purely software TISA. In one such embodiment, the TISA module 2810 is implemented as a purely software

TISA that is configured to access SUT 1910 via a traditional vector JTAG interface. It is noted that use of the deferred scheduler execution architecture 2800 enables such an embodiment, at least because implementation of a purely software TISA with a traditional vector JTAG interface uses a reconstruction of the entire vector from the TISA operations such that it is possible to send the input vector and receive the output vector via the traditional JTAG interface. In this manner, TISA scheduling as depicted and described herein with respect to FIGs. 19 - 27 may be used even where the JTAG TAP is a traditional JTAG TAP (rather than a TAP designed for TISA, which may includes a gated TCK application or a gated TCK TAP controller supporting gating of the TAP TCK signal).

It is noted that the TISA module 2810, the reordering buffer module 2820, and the vector transformation module 2830 may be implemented using hardware and/or software in any other suitable manner.

In a direction toward SUT 1910, TISA module 2810, reordering buffer module 2820, and vector transformation module 2830 are configured to perform functions for enabling the TISA operations received at TISA module 2810 to be applied to SUT as a complete input test vector for the scan chain of SUT 1910. The TISA module 2810 is configured to execute TISA operations in order to determine input test data (e.g., input test data bits) for SUT 1910 and to provide the input test data to reordering buffer module 2820. The reordering buffer module 2820 is configured to receive the input test data from TISA module 2810, buffer the input data in a manner enabling reordering of the input test data to form a complete input test vector for the scan chain of SUT 1910, and provide the input test data of the complete input test vector to vector transformation module 2830. The vector transformation module 2830 is configured to receive the input test data of the complete input test vector from reordering buffer module 2820, transform the input test data of the complete input test vector for use by JTAG TAP 2840, and provide the input

test data (e.g., input test data bits) of the transformed input test vector to the JTAG TAP 2840.

In a direction from SUT 1910, vector transformation module 2830, reordering buffer module 2820, and TISA module 2810 are configured to perform functions for enabling an output test vector received from the scan chain of SUT 1910 to be mapped back to the TISA operations of the associated input test vector applied to the scan chain of SUT 1910. The vector transformation module 2830 is configured to receive an output test vector including output test data (e.g., output test data bits) from JTAG TAP 2840, apply a vector transformation to the output test vector to provide a transformed output test vector including transformed output test data, and provide the transformed output test data of the transformed output test vector to reordering buffer module 2820. The reordering buffer module 2820 is configured to receive the transformed output test data of the transformed output test vector from vector transformation module 2830, preserve the transformed output test data in a buffer representing the transformed output test data to form a complete output vector for the scan chain of SUT 1910, partition the transformed output test data into buffered segments corresponding to positions and lengths of the corresponding segments, and provide the transformed output test data to TISA module 2810. It is noted that each of the test data segments represented in the reordering buffer module 2820 represents scan segments requested by a user application (in this exemplary embodiment, user application 1940). The TISA module 2810 is configured to receive the transformed output test data from reordering buffer module 2820, map the transformed output test data to the TISA operations for which the associated output test data was requested, determine the associated user application(s) performing the respective TISA operations (again, in this exemplary embodiment, user application 1940), and provide test result data to be provided to the user application(s) (e.g., directly where the TISA module 2810 interacts with the user application 1940 directly, or via

the scheduler where the TISA module 2810 interacts with the scheduler 1930 and the scheduler interacts with the user application 1940). In the case in which access requests are issued by multiple user applications and, thus, result in issuance of TISA operations of multiple user applications, the test result data is mapped to the TISA operations of the user applications such that the user applications receive portions of the test result data relevant to the TISA operations that they initiated, respectively.

It is noted that, although omitted for purposes of clarity, the Circuit Model 1934 associated with the SUT 1910 is modified to include the vector transformation module as its root. As depicted in FIG. 21, the exemplary Circuit Model 2120 would include a parent node above the "SUT" node, where the parent node represents the vector transformation module 2830.

It is noted that the use of deferred scheduling, such as via deferred scheduler execution architecture 2800, enables reconstruction of the entire scan chain from its associated scan segments, thereby enabling various types of operations to be performed on the entire scan chain of SUT 1910 (e.g., scrambling, overscan vector operations, and the like, as well as various combinations thereof). For example, in overscan vectors one or more additional bits are added to the scan vector input and, thus, also are being received as part of the scan vector output and must be removed from the scan vector output (i.e., it's not a direct 1:1 relationship in this case), and use of the deferred scheduler execution architecture 2800 enables realignment of the bits of the scan vectors being shifted to account for this type of bit padding.

It is noted that use of deferred scheduling, such as via deferred scheduler execution architecture 2800, enables use of TISA or a TISA-like test instruction set architecture even where gating of TCK is not supported or available.

Although primarily depicted and described herein with respect to an embodiment of a deferred scheduler execution architecture that is a modified

version of the scheduler execution architecture 1900 depicted and described with respect to FIG. 19, it is noted that deferred scheduler execution also may be used with the scheduler execution architecture 2600 of FIG. 26. In this embodiment, deferred scheduling module 2801 would be disposed between scheduler 1930 and JTAG TAP 2840 rather than between user application 1940 and JTAG TAP 2840.

Although omitted for purposes of clarity, it is noted that the various elements of deferred scheduler execution architecture 2800 may be implemented using one or more devices and, similarly, that communication between the various elements of deferred scheduler execution architecture 2800 may be performed in any suitable manner (e.g., depending on the number of devices used, the locations of the device, and the like).

It is noted that various elements of deferred scheduler execution architecture 2800 may be implemented using hardware and/or software. For example, the scheduler 1930 and the user application 1940 each may be implemented in hardware or as a software-based module stored in one or more memories and configured for execution by one or more processors (e.g., TISA module 2810 and/or any other suitable processor(s), which may depend on the locations of TISA module 2810, scheduler 1930, and user application 1940). Many such embodiments related to scheduler 1930, user application 1940, and TISA module 2810 (e.g., TISA processor 1920 in FIG. 19) are depicted and described with respect to FIG. 19. Similarly, for example, as described hereinabove, deferred scheduling module 2801 may be implemented in hardware and/or software (which may include implementation of each of the elements of the deferred scheduling module 2801 in hardware and/or software).

It is noted that the deferred scheduling module 2801 may be co-located with JTAG TAP 2840 and separate from user application 1940, may be co-located with user application 1940 and separate from JTAG TAP 2840, may be co-located with JTAG TAP 2840 and user application 1940 (co-located

with or separate from scheduler 1930), or may be separate from both JTAG TAP 2840 and user application 1940. It is noted that such arrangements are in addition to the various arrangements of TISA module 2810 (e.g., TISA processor 1920), scheduler 1930, and user application 1940 as depicted and described with respect to FIG. 19). In such embodiments, communication between deferred scheduling module 2801 and JTAG TAP 2840, between deferred scheduling module 2801 and user application 1940 (e.g., in the case of deferred scheduler execution architecture 2800 being implemented as a modified version of scheduler execution architecture 1900 in which user application 1940 is configured to provide the TISA operations for the JTAG TAP 2840), and between deferred scheduling module 2801 and scheduler 1930 (e.g., in the case of deferred scheduler execution architecture 2800 being implemented as a modified version of scheduler execution architecture 2600 in which scheduler 1930 is configured to provide the TISA operations for the JTAG TAP 2840)may be provided in any suitable manner (e.g., using direct communication, using communication via one or more communication networks, and the like).

In one embodiment, for example, deferred scheduling module 2801 , scheduler 1930, and user application 1940 may be implemented in separate devices which may communicate with each other via one or more communication networks (e.g., via an Ethernet network, the Internet, and the like, as well as various combinations thereof).

In one embodiment, for example, deferred scheduling module 2801 and scheduler 1930 may be co-located in a single device and user application 1940 may be implemented in a separate device. For example, scheduler 1930 may be a software-based module stored in one or more memories accessible to one or more processors (e.g., TISA module 2810 and/or one or more other processors) configured to execute the scheduler 1930 in order to provide the various functions depicted and described herein. For example, user application 1940 may be implemented in hardware or as a software-

based module stored in one or more memories and configured for execution by one or more processors to provide the various functions depicted and described herein. For example, communication between scheduler 1930 and user application 1940 may be via one or more communication networks (e.g., via an Ethernet network, the Internet, and the like, as well as various combinations thereof).

In one embodiment, for example, deferred scheduling module 2801 and user application 1940 may be co-located in a single device and scheduler 1930 may be implemented in a separate device. For example, user application 1940 may be a software-based module stored in one or more memories accessible to one or more processors (e.g., TISA module 2810 and/or one or more other processors) configured to execute the user application 1940 in order to provide the various functions depicted and described herein. For example, scheduler 1930 may be implemented in hardware or as a software-based module stored in one or more memories and configured for being executed by one or more processors to provide the various functions depicted and described herein. For example, communication between user application 1940 and scheduler 1930 may be via one or more communication networks (e.g., via an Ethernet network, the Internet, and the like, as well as various combinations thereof).

In one embodiment, for example, scheduler 1930 and user application 1940 may be co-located in a single device and deferred scheduling module 2801 may be implemented as a separate device. For example, scheduler 1930 and user application 1940 each may be implemented in hardware or as a software-based module stored in one or more memories and configured to be executed by one or more processors to provide the various functions depicted and described herein. For example, communication between user application 1940 and TISA module 2810 may be via one or more communication networks (e.g., via an Ethernet network, the Internet, and the like, as well as various combinations thereof).

In one embodiment, for example, deferred scheduling module 2801, scheduler 1930, and user application 1940 may be co-located in a single device. For example, TISA processor 1920, scheduler 1930, and user application 1940 may be software-based modules stored in one or more memories executing on one or more processors. For example, scheduler 1930 and user application 1940 may be software-based modules stored in one or more memories accessible to one or more processors which may access and execute the scheduler 1930 and user application 1940 to provide the various functions depicted and described herein. For example, TISA module 2810 may be an application running a library of ISA instructions that emulate the operation of a TISA processor (e.g., the TISA processor 1920) whereby actual writes and reads to the SUT are deferred by placing the vector data into the reordering buffer module 2820 to later be applied as a unified vector to the SUT. An exemplary embodiment is depicted and described with respect to FIG. 29.

FIG. 29 depicts one embodiment of a deferred scheduler execution architecture configured to provide position-based scheduling for testing a system under test.

As depicted in FIG. 29, deferred scheduler execution architecture 2900 is similar to deferred scheduler execution architecture 2800 of FIG. 28 (and also is a modified version of scheduler execution architecture 2300 of FIG. 23 which depicts an exemplary embodiment of scheduler execution architecture 1900 of FIG. 19).

The deferred scheduler execution architecture 2900 includes SUT 2310, Main CPU 2320, and memory 2324 of scheduler execution architecture 2300 of FIG. 23, including modifications to the Main CPU 2320 and memory 2324 to support an embodiment of deferred scheduler execution architecture 2800 of FIG. 28 (illustratively, an exemplary embodiment in which deferred scheduler execution architecture 2800 is implemented in software).

The memory 2324, in addition to including scheduler 2330 and user application 2340 of scheduler execution architecture 2300 of FIG. 23, also includes a deferred scheduling module 2901 . The deferred scheduling module 2901 includes a TISA module 291 0, a reordering buffer module 2920, and a vector transformation module 2930 which are software-based implementations of the TISA module 281 0, reordering buffer module 2820, and vector transformation module 2830 of deferred scheduler execution architecture 2800 of FIG. 28, respectively.

The Main CPU 2320, rather than including TISA 2322 of the scheduler execution architecture 2300 of FIG. 23 (since TISA processing is provided by TISA module 291 0 of deferred scheduling module 2901 of memory 2324), includes a JTAG TAP 2940 corresponding to JTAG TAP 2840 of deferred scheduler execution architecture 2800 of FIG. 28.

Although FIG. 29 is primarily depicted and described herein with respect to an embodiment in which the deferred scheduling capability is implemented within software, it will be appreciated (as depicted and described with respect to deferred scheduler execution architecture 2800 of FIG. 28), that the deferred scheduling capability may be implemented using hardware and/or software. In one embodiment, for example, vector transformation module 2930 may be implemented as a hardware module outside of memory 2324, while reordering buffer 2920 and TISA module 291 0 may still be implemented using software and, thus, may still be stored within memory 2324. In one embodiment, for example, vector transformation module 2930 and reordering buffer 2920 may be implemented as one or more hardware modules outside of memory 2324, while TISA module 291 0 may still be implemented using software and, thus, may still be stored within memory 2324. Various other arrangements are contemplated.

Although primarily depicted and described herein with respect to embodiments in which the deferred scheduler execution architecture is implemented based on the scheduler execution architecture 1900 of FIG. 19

(in which the user application 1940 interacts with the TISA processor 1920), it is noted that the deferred scheduler execution architecture also may be implemented based on the scheduler execution architecture 2600 of FIG. 26 (in which the scheduler 1930 interacts with the TISA processor 1920 / TISA module 2810). This latter solution has the advantage of presenting a single point of origin for TISA operations (namely, scheduler 1930), making interfacing and synchronisation with TISA module 2810 or TISA module 2910 much easier.

Although primarily depicted and described herein with respect to embodiments in which the deferred scheduling capability is provided in conjunction with scheduler execution architectures and environments that are depicted and described herein, it is noted that the deferred scheduling capability may be provided in conjunction with any other suitable type of scheduling-based testing architectures and environments.

In many of the embodiments of position-based scheduling that are depicted and described herein, an assumption is made that each of the scan segments of the scan chain of the SUT has an application assigned thereto, such that the applications(s) provide test data sufficient to populate the global scan chain. In many cases, however, one or more of the segments of which the scan chain is composed may not have any application assigned thereto when tests are executed using position-based scheduling. In these cases, data still needs to be provided for these unused segments of the scan chain. In other words, if no application is specifically asking to update the segment and the segment is in series with the allocated segments of applications, the global scan chain scan still needs to have data associated with those gaps in the scan chain. This may be handled in a number of ways.

In one embodiment, for example, the scheduler includes (or at least has access to) default data about each segment of the scan chain (e.g., TISA operations) such that it can provide data to fill any unused segments.

In one embodiment, for example, an application that is configured to deal with unallocated scan segments of the scan chain coordinates with the scheduler to provide data for the unallocated segments such that a complete scan vector can be generated. It is noted that supporting such a capability for providing data for each segment of the scan chain (including for unallocated segments) is particularly relevant in the case of a TISA-based implementation, because even though TISA allows any given application to focus only on a subset of the scan chain rather than the entire scan chain (e.g., on one or more target segments of which the scan chain is composed), the 1149.1 JTAG standard still requires that all segments of the scan chain be scanned in order to complete the entire scan sequence. In one embodiment, it may be included as part of an acknowledgement from the scheduler since the scheduler knows the topology of the entire scan chain.

Although primarily depicted and described herein within the context of providing position-based scheduling in a testing architecture using TISA (and associated TISA operations), it is noted that various embodiments of position-based scheduling may be provided in testing architectures using other types of test instruction sets (and other types of associated test operations). Similarly, although primarily depicted and described herein within the context of providing deferred scheduling in a testing architecture using TISA (and associated TISA operations), it is noted that various embodiments of deferred scheduling may be provided in testing architectures using other types of test instruction sets (and other types of associated test operations).

FIG. 30 depicts a high-level block diagram of a computer suitable for use in performing the functions described herein. As depicted in FIG. 30, computer 3000 includes a processor element 3002 (e.g., a central processing unit (CPU) or other suitable processor(s)), a memory 3004 (e.g., random access memory (RAM), read only memory (ROM), and/or any other suitable types of memory), system testing module/process 3005 adapted for performing system testing functions depicted and described herein, and

various input/output devices 3006 (e.g., a user input device (such as a keyboard, a keypad, a mouse, and the like), a user output device (such as a display, a speaker, and the like), an input port, an output port, a receiver, a transmitter, and storage devices (e.g., a tape drive, a floppy drive, a hard disk drive, a compact disk drive, and the like)).

It should be noted that system testing functions depicted and described herein may be implemented in software and/or in a combination of software and hardware, e.g., using a general purpose computer, one or more application specific integrated circuits (ASIC), and/or any other hardware equivalents. In one embodiment, system testing process 3005 can be loaded into memory 3004 and executed by processor 3002 to implement and/or support implementation of at least a portion of the system testing functions described hereinabove. Thus, system testing process 3005 (including associated data structures) can be stored on a computer readable storage medium or carrier, e.g., RAM memory, magnetic or optical drive or diskette, and the like.

It is contemplated that some of the steps discussed herein as software methods may be implemented within hardware, for example, as circuitry that cooperates with the processor to perform various method steps. Portions of the functions/elements described herein may be implemented as a computer program product wherein computer instructions, when processed by a computer, adapt the operation of the computer such that the methods and/or techniques described herein are invoked or otherwise provided. Instructions for invoking the inventive methods may be stored in fixed or removable media, transmitted via a data stream in a broadcast or other signal bearing medium, and/or stored within a memory within a computing device operating according to the instructions.

Aspects of various embodiments are specified in the claims. Those and other aspects of various embodiments are specified in the following numbered clauses:

1. An apparatus, comprising:

a processing module configured to receive a plurality of test operations associated with a plurality of segments of a unit under test and to generate therefrom input test data configured to be applied to the unit under test via a Test Access Port (TAP);

a reordering buffer module configured to receive the input test data from the processing element and to buffer the input test data in a manner for reordering the input test data to compose an input test vector for a scan chain of the unit under test; and

a vector transformation module configured to receive the input test vector from the reordering buffer module and to apply a vector transformation for the input test vector.

2. The apparatus of clause 1, wherein the processing module is configured to receive at least a portion of the test operations from one of:

an application configured to generate test operations;

a scheduler configured to schedule test operations; and

a testing tool.

3. The apparatus of clause 1, wherein the processing module is configured to receive the test operations via a direct connection or a communication network.

4. The apparatus of clause 1, further comprising:

a JTAG TAP configured to interface with the scan chain of the unit under test;

wherein the vector transformation module is configured to provide the transformed input test vector to the JTAG TAP.

5. The apparatus of clause 1, further comprising a processor and a memory, wherein the memory is configured to store at least one of the processing module, the reordering buffer module, and the vector transformation module.

6. The apparatus of clause 1, wherein the vector transformation module is further configured to:

receive an output test vector of the unit under test, the output test vector comprising output test data;

apply a vector transformation to the output test vector to provide transformed output test data; and

propagate the transformed output test data toward the reordering buffer module.

7. The apparatus of clause 6, wherein the reordering buffer module is further configured to:

receive the transformed output test data from the vector transformation module;

preserve the transformed output test data in a buffer representing the transformed output test data to form a complete output test vector for the scan chain of the unit under test;

partition the transformed output test data into buffered segments corresponding to positions and lengths of the respective scan chain segments; and

propagate the transformed output test data toward the processing module.

8. The apparatus of clause 7, wherein the processing module is further configured to:

receive the transformed output test data from the reordering buffer module;

map the transformed output test data to test operations for which the associated output test data was requested;

determine one or more user applications associated with the test operations; and

provide test result data for the one or more user applications associated with the test operations.

9. The apparatus of clause 8, wherein the processing module is further configured to:

propagate the test result data toward the one or more user applications with which the test result data is associated.

10. The apparatus of clause 8, wherein the processing module is further configured to:

propagate the test result data toward a scheduler from which the test operations were received by the processing module.

11. A non-transitory computer-readable storage medium storing instruction which, when executed by a processor, cause the processor to perform a method, the method comprising:

receiving a plurality of test operations associated with a plurality of segments of a unit under test;

generating, from the test operations, input test data configured to be applied to the unit under test via a Test Access Port (TAP);

buffering the input test data in a manner for reordering the input test data to compose an input test vector for a scan chain of the unit under test; and

applying a vector transformation for the input test vector.

12. The non-transitory computer-readable storage medium of clause 11, wherein at least a portion of the test operations are received from one of:

an application configured to generate test operations;

a scheduler configured to schedule test operations; and

a testing tool.

13. The non-transitory computer-readable storage medium of clause 11, wherein the test operations are received via a direction connection or a communication network.

14. The non-transitory computer-readable storage medium of clause 11, the method further comprising:

providing the transformed input test vector to a JTAG TAP configured to interface with the scan chain of the unit under test.

15. The non-transitory computer-readable storage medium of clause 11, the method further comprising:

receiving an output test vector of the unit under test, the output test vector comprising output test data; and

applying a vector transformation to the output test vector to provide transformed output test data.

16. The non-transitory computer-readable storage medium of clause 15, the method further comprising:

receiving the transformed output test data;

preserving the transformed output test data in a buffer representing the transformed output test data to form a complete output test vector for the scan chain of the unit under test; and

partitioning the transformed output test data into buffered segments corresponding to positions and lengths of the respective scan chain segments.

17. The non-transitory computer-readable storage medium of clause 16, the method further comprising:

mapping the transformed output test data to test operations for which the associated output test data was requested;

determining one or more user applications associated with the test operations; and

providing test result data for the one or more user applications associated with the test operations.

18. The non-transitory computer-readable storage medium of clause 17, the method further comprising:

propagating the test result data toward the one or more user applications with which the test result data is associated.

19. The non-transitory computer-readable storage medium of clause 17, the method further comprising:

propagating the test result data toward a scheduler from which the test operations were received.

20. A method, comprising:

receiving a plurality of test operations associated with a plurality of segments of a unit under test;

generating, from the test operations, input test data configured to be applied to the unit under test via a Test Access Port (TAP);

buffering the input test data in a manner for reordering the input test data to compose an input test vector for a scan chain of the unit under test; and

applying a vector transformation for the input test vector.

Although various embodiments which incorporate the teachings of the present invention have been shown and described in detail herein, those skilled in the art can readily devise many other varied embodiments that still incorporate these teachings.

What is claimed is:

1. An apparatus, comprising:

a processing module configured to receive a plurality of test operations associated with a plurality of segments of a unit under test and to generate therefrom input test data configured to be applied to the unit under test via a Test Access Port (TAP);

a reordering buffer module configured to receive the input test data from the processing element and to buffer the input test data in a manner for reordering the input test data to compose an input test vector for a scan chain of the unit under test; and

a vector transformation module configured to receive the input test vector from the reordering buffer module and to apply a vector transformation for the input test vector.

2. The apparatus of claim 1, wherein the processing module is configured to receive at least a portion of the test operations from one of:

an application configured to generate test operations;

a scheduler configured to schedule test operations; and

a testing tool.

3. The apparatus of claim 1, further comprising:

a JTAG TAP configured to interface with the scan chain of the unit under test;

wherein the vector transformation module is configured to provide the transformed input test vector to the JTAG TAP.

4. The apparatus of claim 1, further comprising a processor and a memory, wherein the memory is configured to store at least one of the

processing module, the reordering buffer module, and the vector transformation module.

5. The apparatus of claim 1, wherein the vector transformation module is further configured to:

receive an output test vector of the unit under test, the output test vector comprising output test data;

apply a vector transformation to the output test vector to provide transformed output test data; and

propagate the transformed output test data toward the reordering buffer module.

6. The apparatus of claim 5, wherein the reordering buffer module is further configured to:

receive the transformed output test data from the vector transformation module;

preserve the transformed output test data in a buffer representing the transformed output test data to form a complete output test vector for the scan chain of the unit under test;

partition the transformed output test data into buffered segments corresponding to positions and lengths of the respective scan chain segments; and

propagate the transformed output test data toward the processing module.

7. The apparatus of claim 6, wherein the processing module is further configured to:

receive the transformed output test data from the reordering buffer module;

map the transformed output test data to test operations for which the associated output test data was requested;

determine one or more user applications associated with the test operations; and

provide test result data for the one or more user applications associated with the test operations.

8. The apparatus of claim 7, wherein the processing module is further configured for at least one of:

propagating the test result data toward the one or more user applications with which the test result data is associated; or

propagating the test result data toward a scheduler from which the test operations were received by the processing module.

9. A non-transitory computer-readable storage medium storing instruction which, when executed by a processor, cause the processor to perform a method, the method comprising:

receiving a plurality of test operations associated with a plurality of segments of a unit under test;

generating, from the test operations, input test data configured to be applied to the unit under test via a Test Access Port (TAP);

buffering the input test data in a manner for reordering the input test data to compose an input test vector for a scan chain of the unit under test; and

applying a vector transformation for the input test vector.

10. A method, comprising:

receiving a plurality of test operations associated with a plurality of segments of a unit under test;

generating, from the test operations, input test data configured to be applied to the unit under test via a Test Access Port (TAP);

buffering the input test data in a manner for reordering the input test data to compose an input test vector for a scan chain of the unit under test; and

applying a vector transformation for the input test vector.

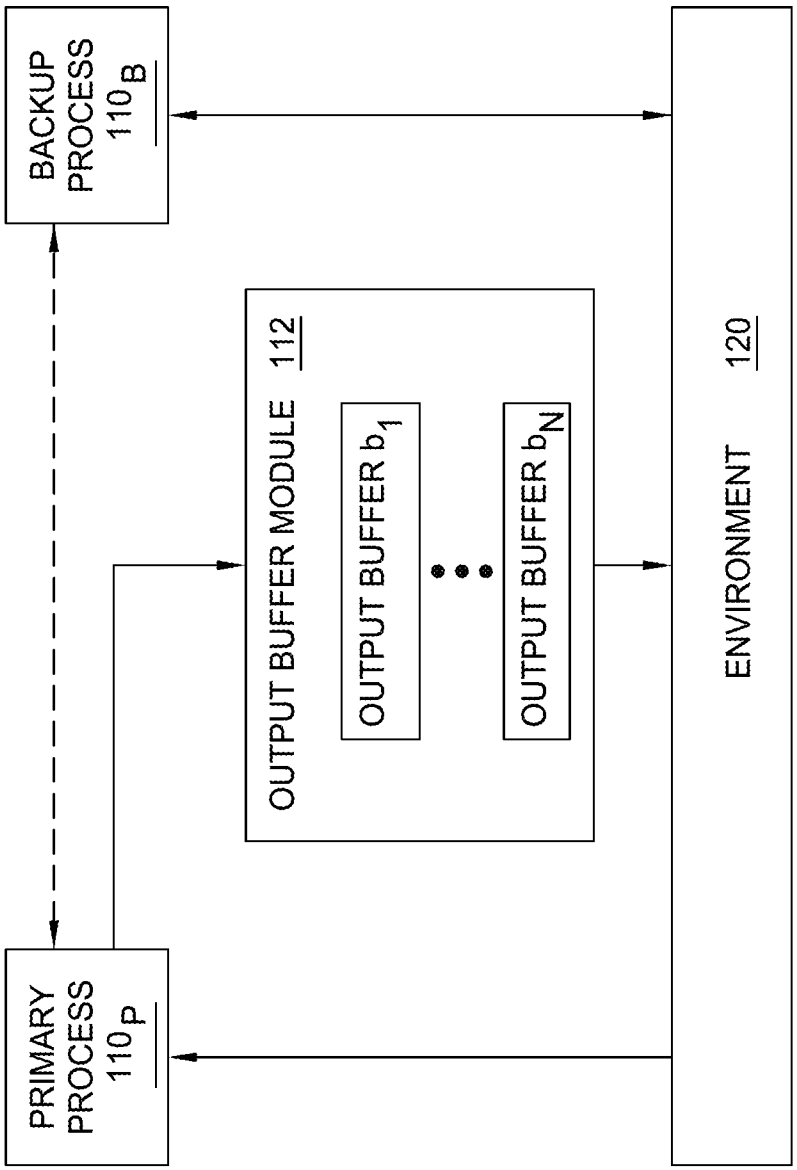
BACKUP PROCESS 110B
PRIMARY PROCESS 110P
OUTPUT BUFFER MODULE 112
OUTPUT BUFFER b1
OUTPUT BUFFER bN
ENVIRONMENT 120

FIG. 1

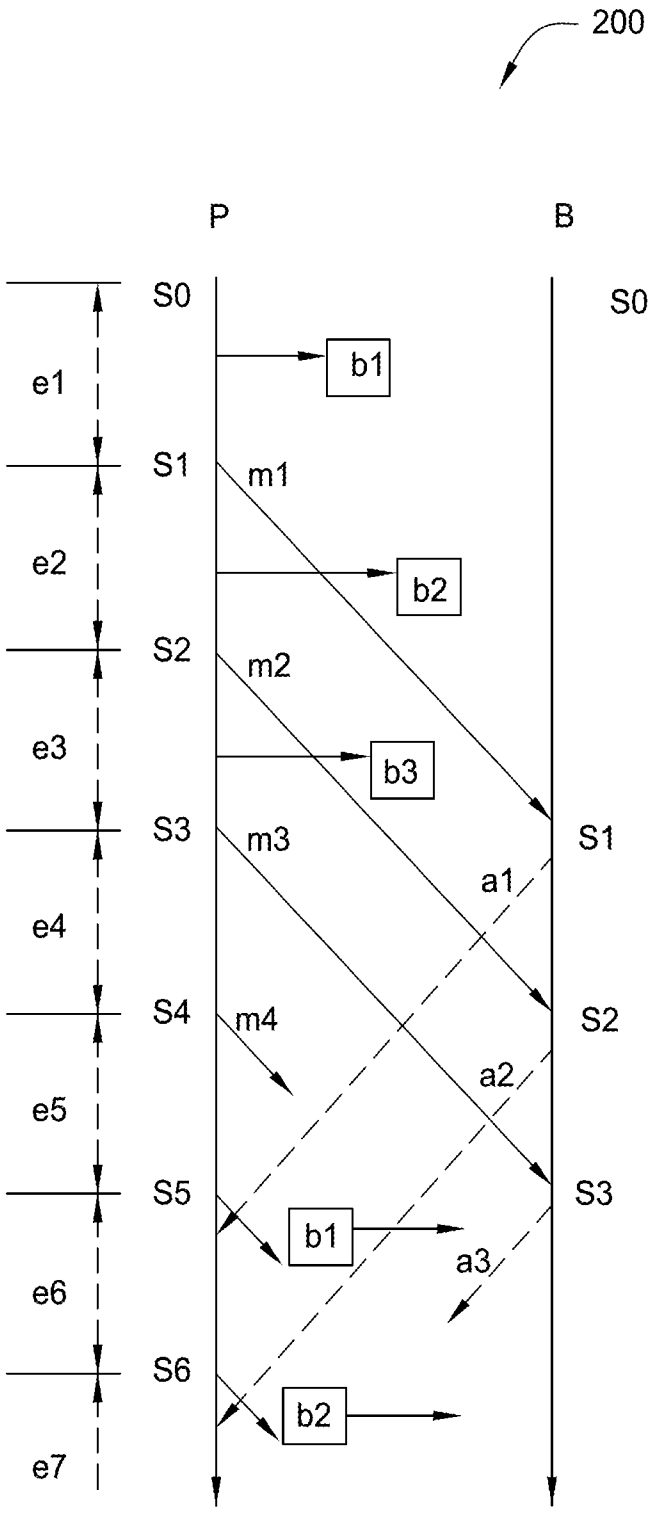
200
P
B
S0
S0
e1
b1
S1
m1
e2
b2
S2
m2
e3
b3
S3
m3
S1
a1
e4
S4
m4
S2
a2
e5
S5
b1
S3
a3
e6
S6
b2
e7

FIG. 2

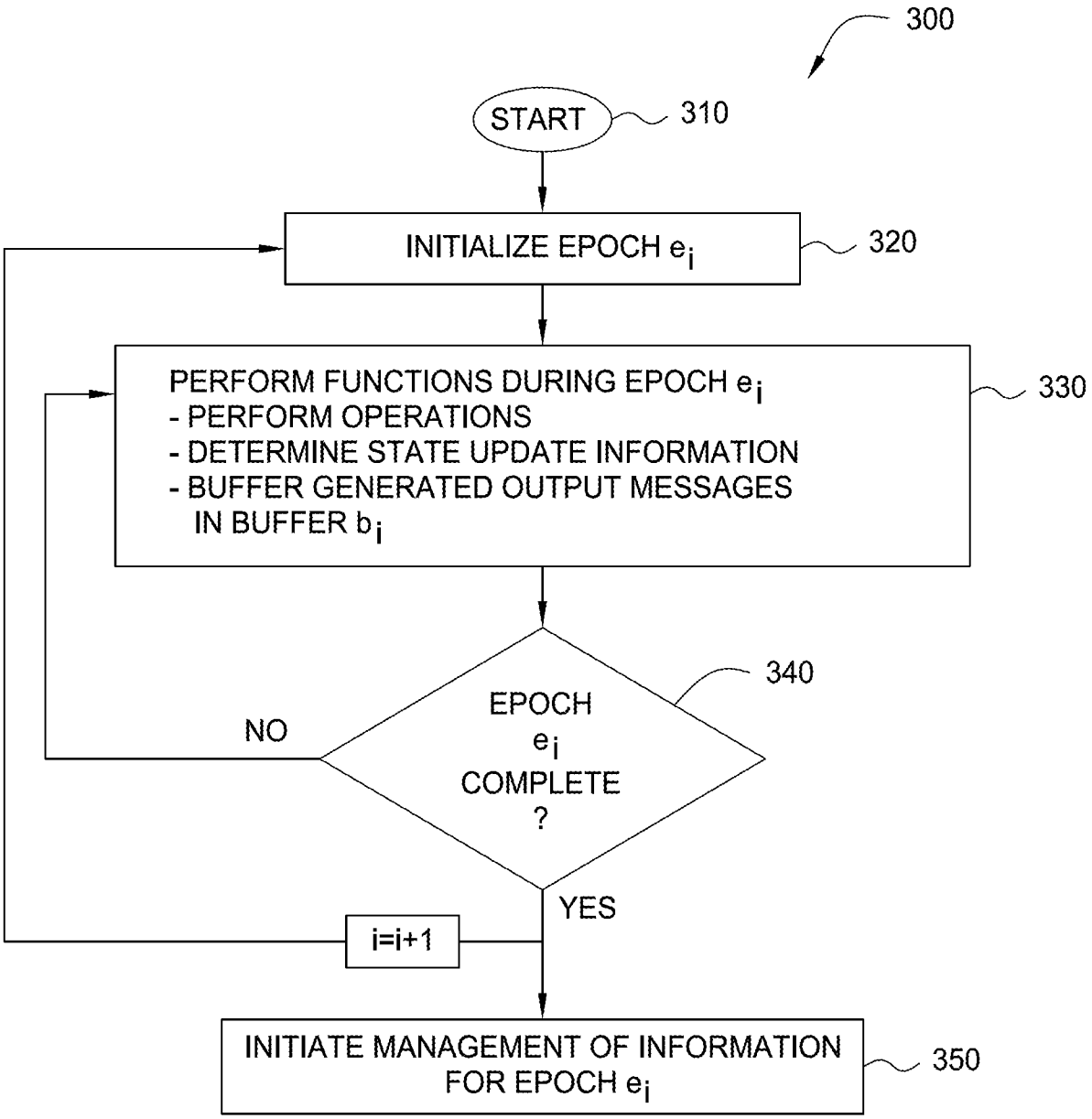
300
START
310
INITIALIZE EPOCH e_i
320
PERFORM FUNCTIONS DURING EPOCH e_i
- PERFORM OPERATIONS
- DETERMINE STATE UPDATE INFORMATION
- BUFFER GENERATED OUTPUT MESSAGES IN BUFFER b_i
330
EPOCH e_i COMPLETE ?
340
NO
YES
i=i+1
INITIATE MANAGEMENT OF INFORMATION FOR EPOCH e_i
350

FIG. 3

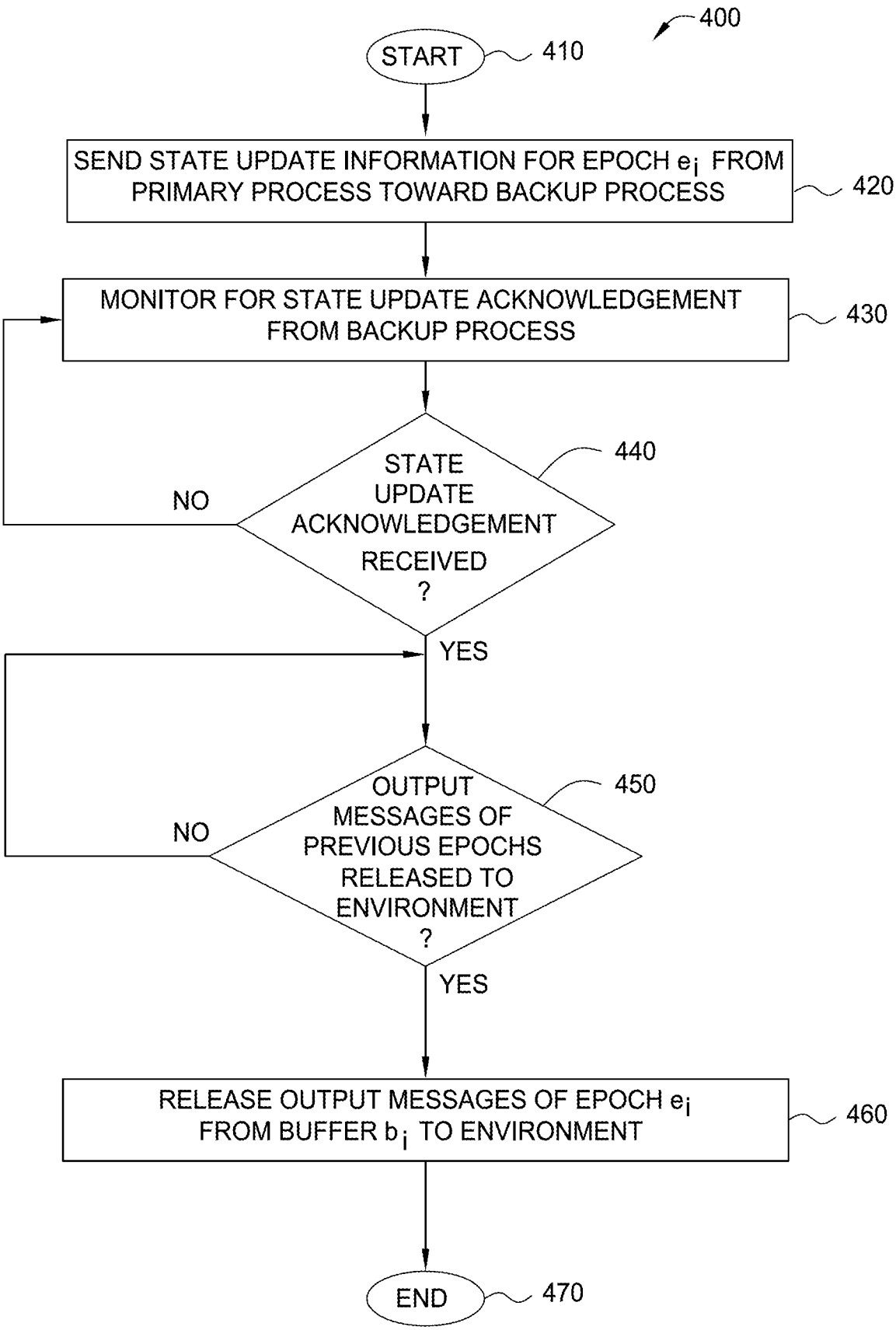
400
START
410
SEND STATE UPDATE INFORMATION FOR EPOCH e_i FROM PRIMARY PROCESS TOWARD BACKUP PROCESS
420
MONITOR FOR STATE UPDATE ACKNOWLEDGEMENT FROM BACKUP PROCESS
430
STATE UPDATE ACKNOWLEDGEMENT RECEIVED ?
440
NO
YES
OUTPUT MESSAGES OF PREVIOUS EPOCHS RELEASED TO ENVIRONMENT ?
450
NO
YES
RELEASE OUTPUT MESSAGES OF EPOCH e_i FROM BUFFER b_i TO ENVIRONMENT
460
END
470

FIG. 4

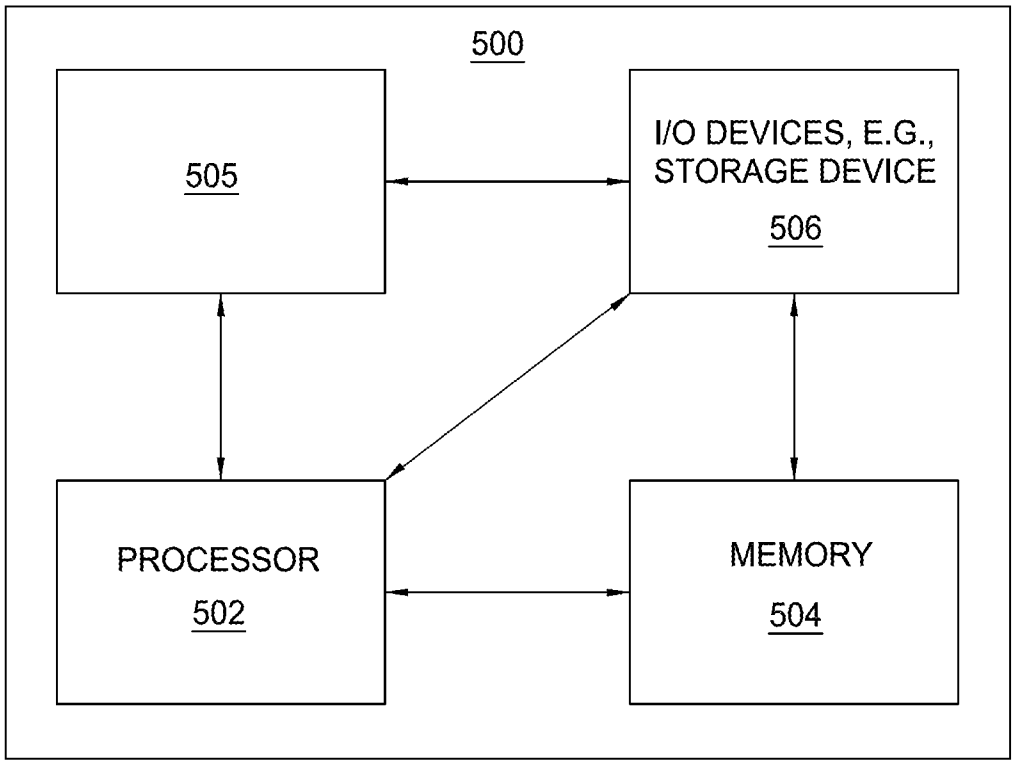
500
505
I/O DEVICES, E.G., STORAGE DEVICE
506
PROCESSOR
502
MEMORY
504

FIG. 5

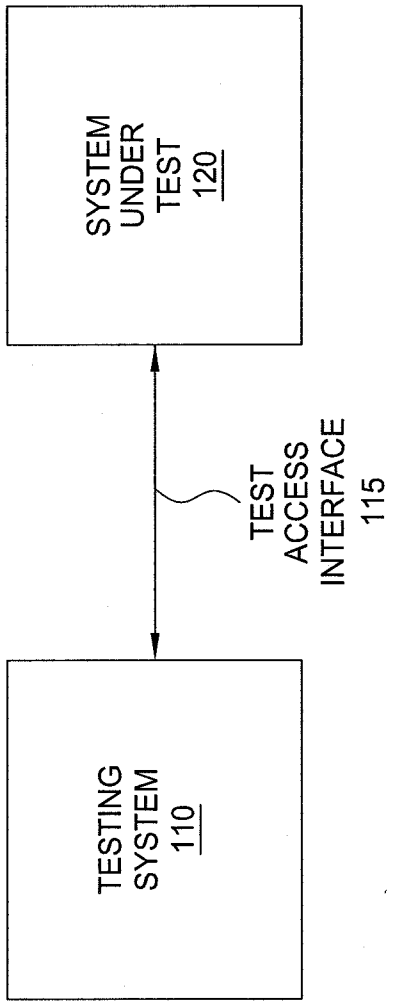

FIG. 1

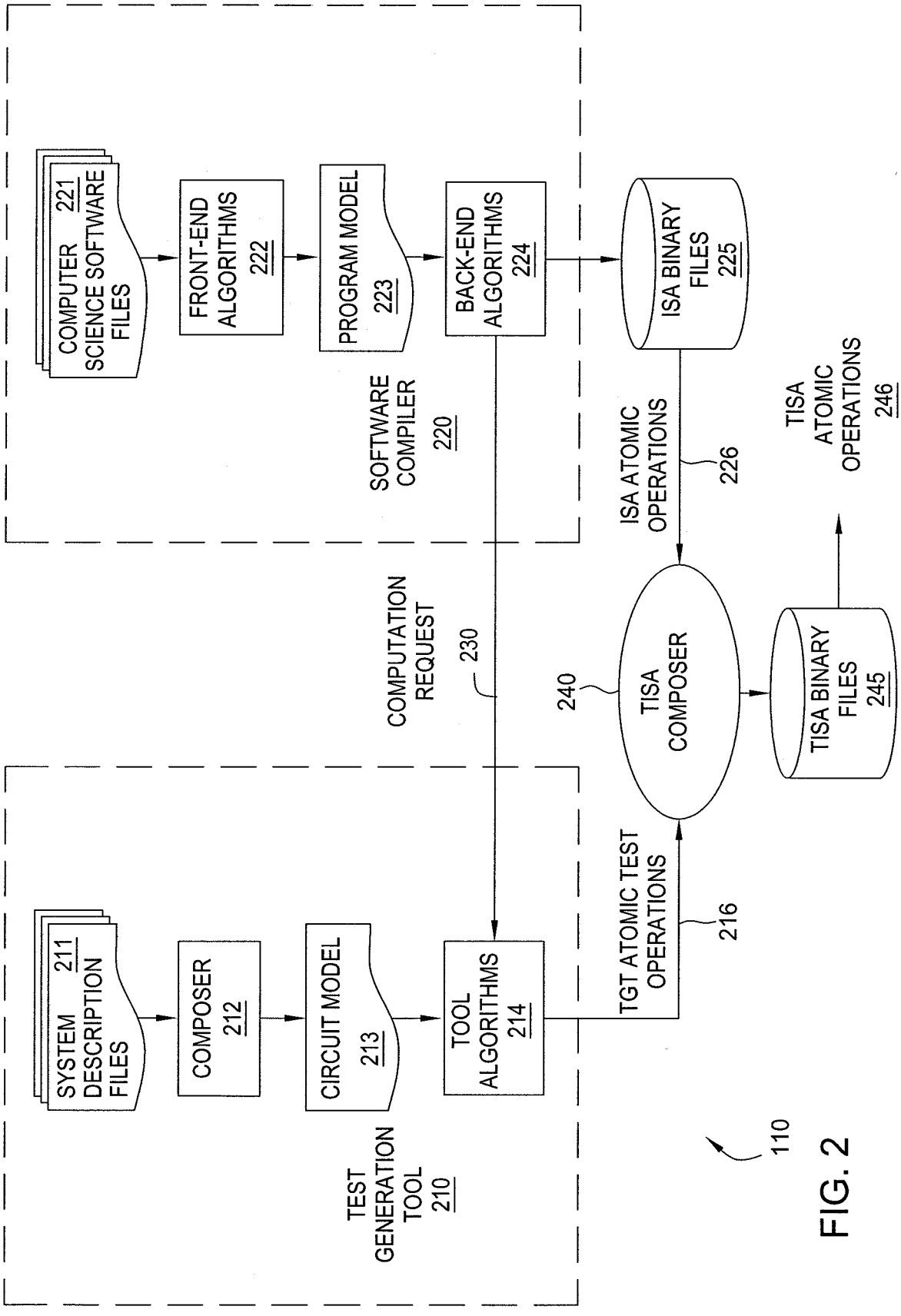
COMPUTER SCIENCE SOFTWARE FILES 221
FRONT-END ALGORITHMS 222
PROGRAM MODEL 223
BACK-END ALGORITHMS 224
ISA BINARY FILES 225
SOFTWARE COMPILER 220
ISA ATOMIC OPERATIONS 226
TISA ATOMIC OPERATIONS 246
COMPUTATION REQUEST 230
240
TISA COMPOSER
TISA BINARY FILES 245
SYSTEM DESCRIPTION FILES 211
COMPOSER 212
CIRCUIT MODEL 213
TOOL ALGORITHMS 214
TGT ATOMIC TEST OPERATIONS 216
TEST GENERATION TOOL 210
110

FIG. 2

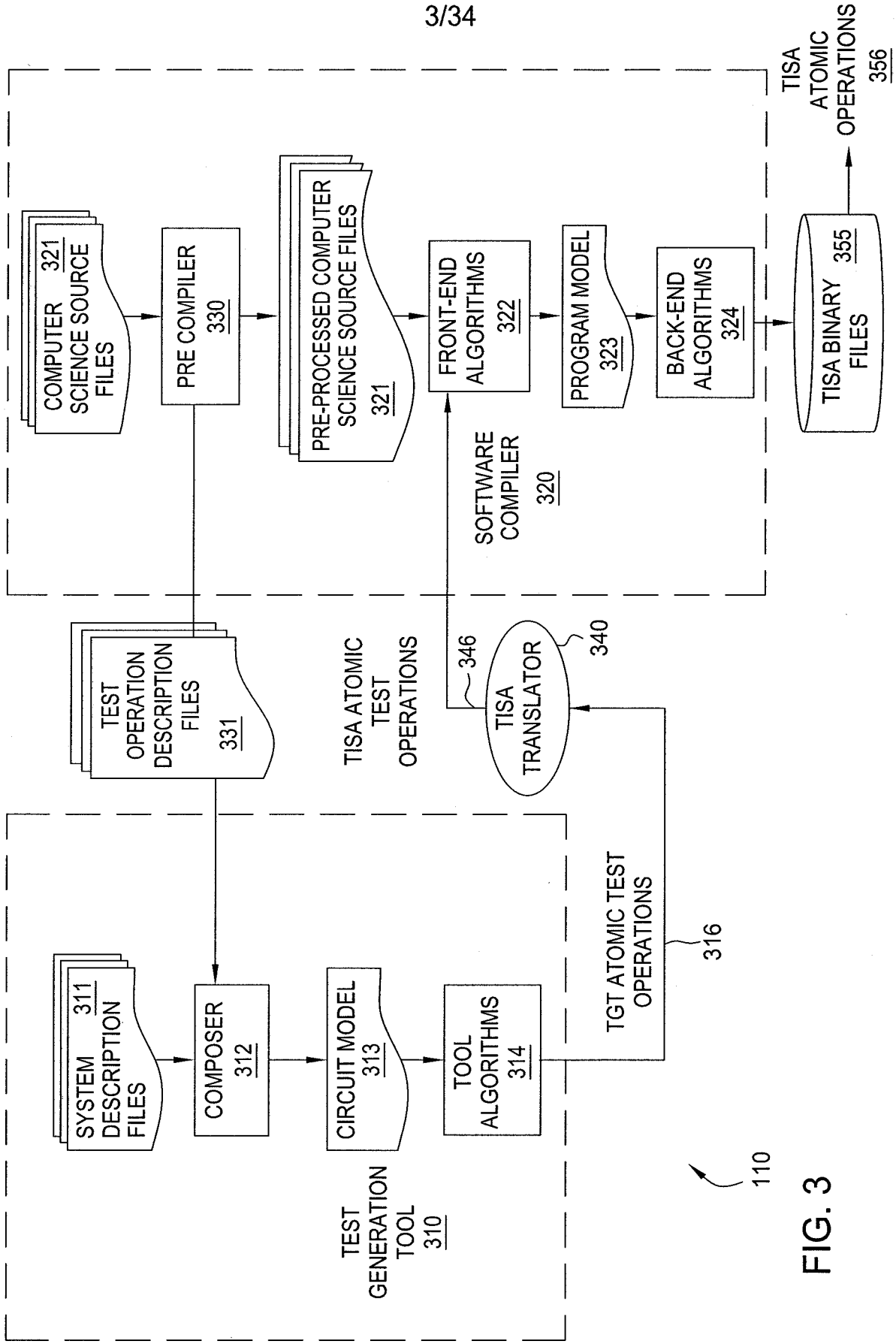
TISA ATOMIC OPERATIONS 356
COMPUTER SCIENCE SOURCE FILES 321
PRE COMPILER 330
PRE-PROCESSED COMPUTER SCIENCE SOURCE FILES 321
FRONT-END ALGORITHMS 322
PROGRAM MODEL 323
BACK-END ALGORITHMS 324
TISA BINARY FILES 355
SOFTWARE COMPILER 320
TEST OPERATION DESCRIPTION FILES 331
TISA ATOMIC TEST OPERATIONS
346
TISA TRANSLATOR
340
SYSTEM DESCRIPTION FILES 311
COMPOSER 312
CIRCUIT MODEL 313
TOOL ALGORITHMS 314
TGT ATOMIC TEST OPERATIONS
316
TEST GENERATION TOOL 310
110

FIG. 3

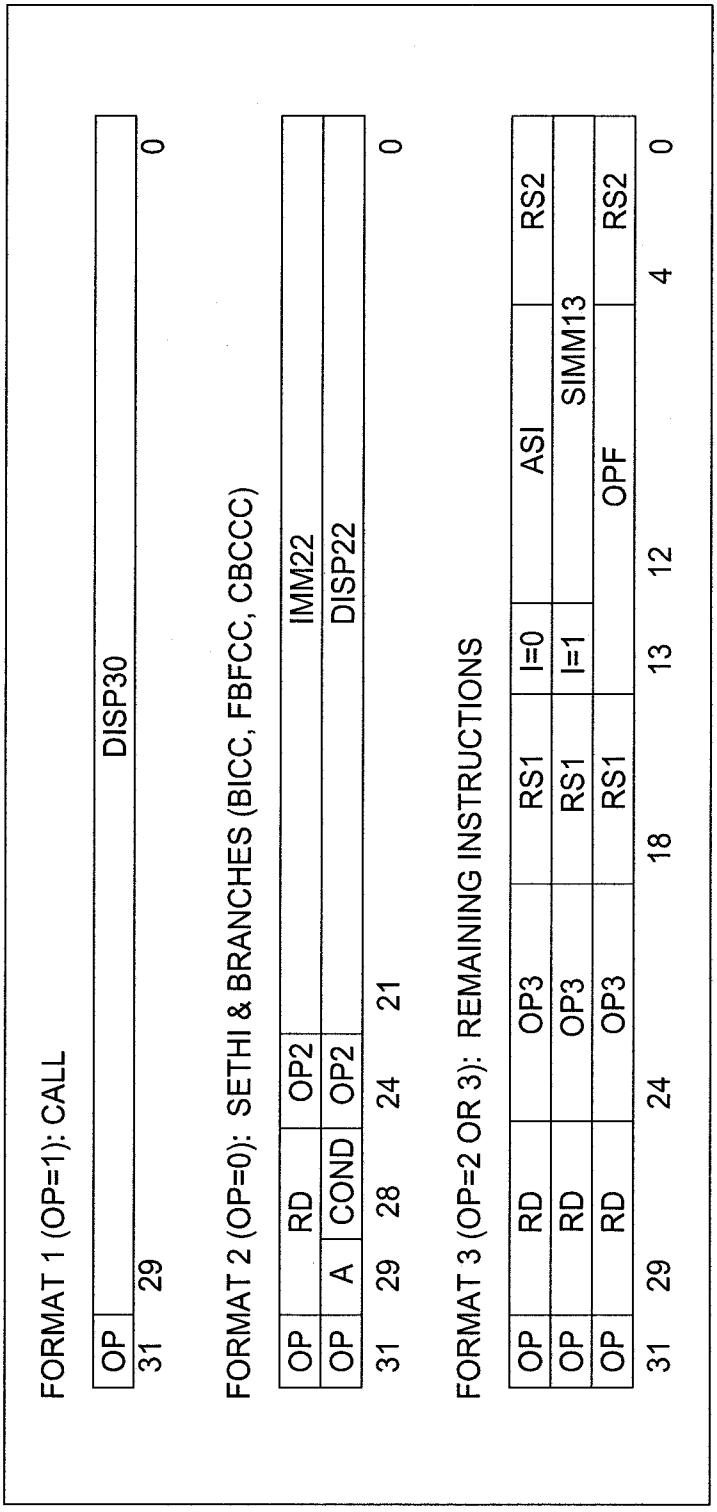

FIG. 4A

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14
0	SVF CODING						0	0	0	TAP STATE				O1	O2	O3	O4

FIG. 4B

IEEE 1149.1 TAP STATE NAME	SVF TAP STATE NAME	4-BIT CODING
Test-Logic-Reset	RESET	0000
Run-Test/Idle	IDLE	0001
Select-DR-Scan	DRSELECT	0010
Capture-DR	DRCAPTURE	0011
Shift-DR	DRSHIFT	0100
Exit1-DR	DREXIT1	0101
Pause-DR	DRPAUSE	0110
Exit2-DR	DREXIT2	0111
Update-DR	DRUPDATE	1000
Select-IR-Scan	IRSELECT	1001
Capture-IR	IRCAPTURE	1010
Shift-IR	IRSHIFT	1011
Exit1-IR	IREXIT1	1100
Pause-IR	IRPAUSE	1101
Exit2-IR	IREXIT2	1110
Update-IR	IRUPDATE	1111

FIG. 4C

TDI
TDO
MASK
SMASK
GENERIC1
GENERIC2

FIG. 4D

INSTRUCTION	SVF CODING
ENDDR	000000
ENDIR	000001
STATE	000010
FREQUENCY	000011
PIO	000100
HDR	000110
HIR	000111
SDR	001000
SIR	001001
TDR	001010
TIR	001011

FIG. 4E

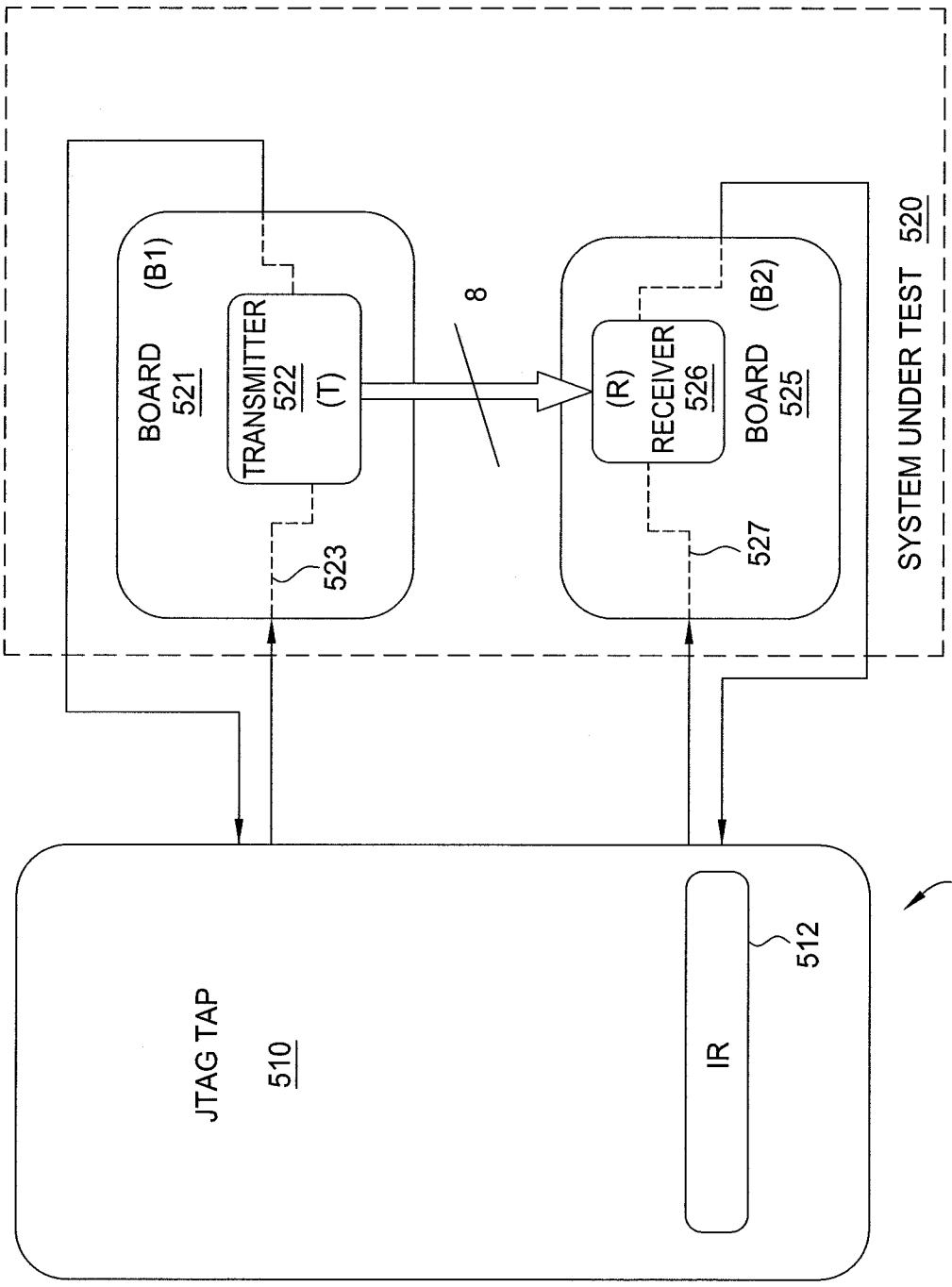
SYSTEM UNDER TEST 520
BOARD (B1) 521
TRANSMITTER 522 (T)
523
8
(R) RECEIVER 526
BOARD (B2) 525
527
JTAG TAP 510
IR
512
500

FIG. 5A

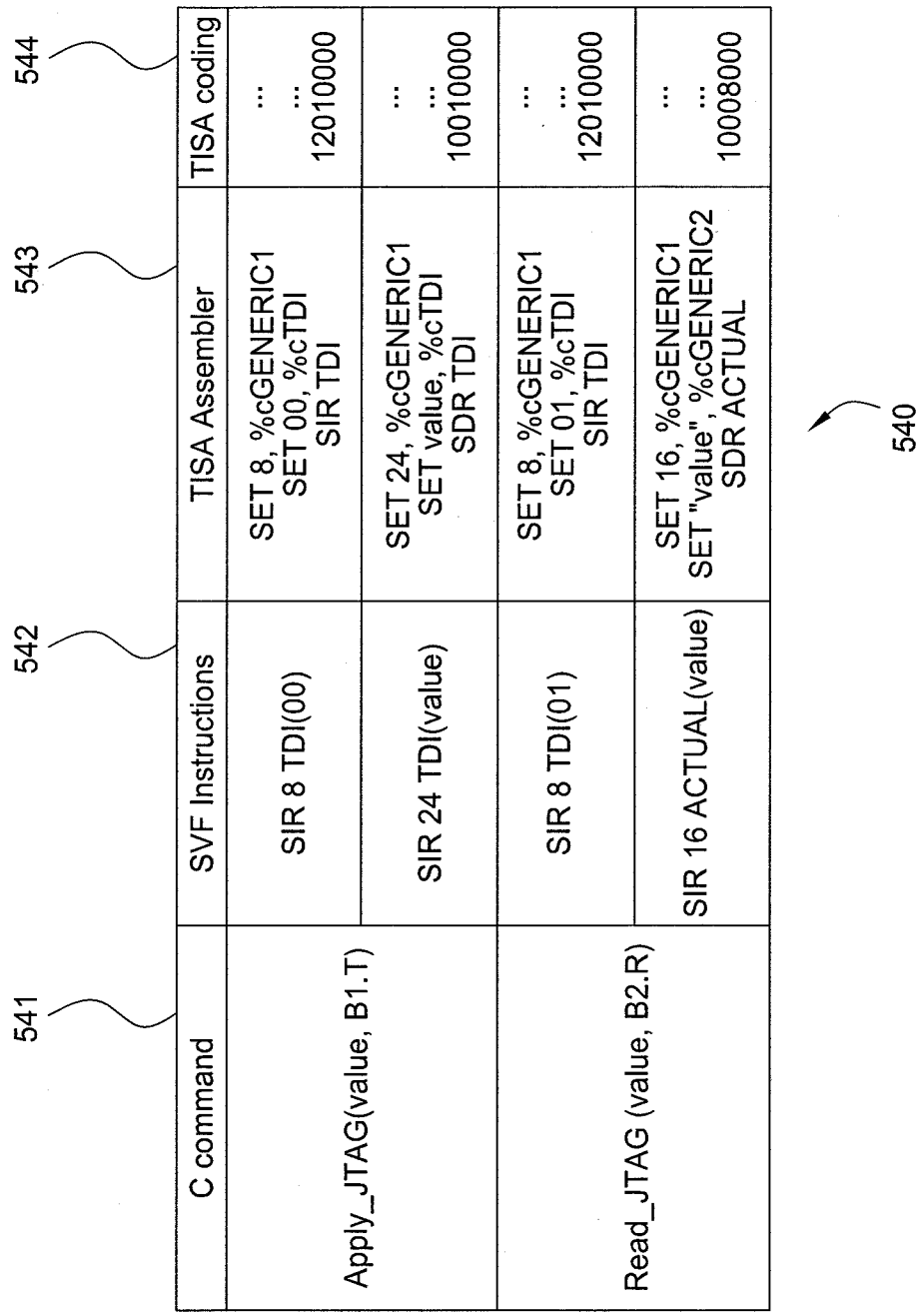
541
542
543
544
540
C command
SVF Instructions
TISA Assembler
TISA coding
Apply_JTAG(value, B1.T)
SIR 8 TDI(00)
SET 8, %cGENERIC1
SET 00, %cTDI
SIR TDI
...
...
12010000
SIR 24 TDI(value)
SET 24, %cGENERIC1
SET value, %cTDI
SDR TDI
...
...
10010000
Read_JTAG (value, B2.R)
SIR 8 TDI(01)
SET 8, %cGENERIC1
SET 01, %cTDI
SIR TDI
...
...
12010000
SIR 16 ACTUAL(value)
SET 16, %cGENERIC1
SET "value", %cGENERIC2
SDR ACTUAL
...
...
10008000

FIG. 5B

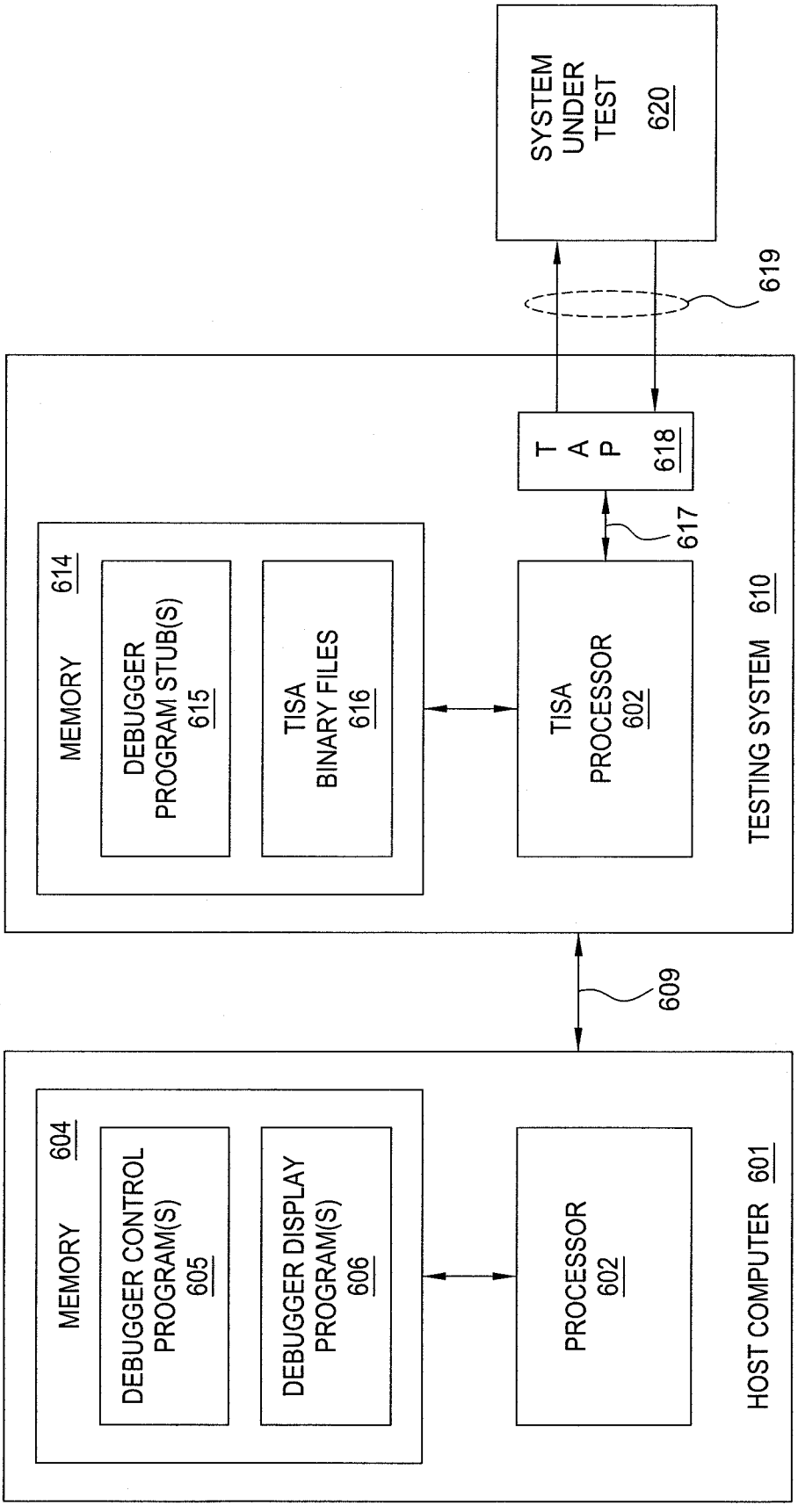
SYSTEM UNDER TEST 620
619
TAP 618
617
600
MEMORY 614
DEBUGGER PROGRAM STUB(S) 615
TISA BINARY FILES 616
TISA PROCESSOR 602
TESTING SYSTEM 610
609
MEMORY 604
DEBUGGER CONTROL PROGRAM(S) 605
DEBUGGER DISPLAY PROGRAM(S) 606
PROCESSOR 602
HOST COMPUTER 601

FIG. 6

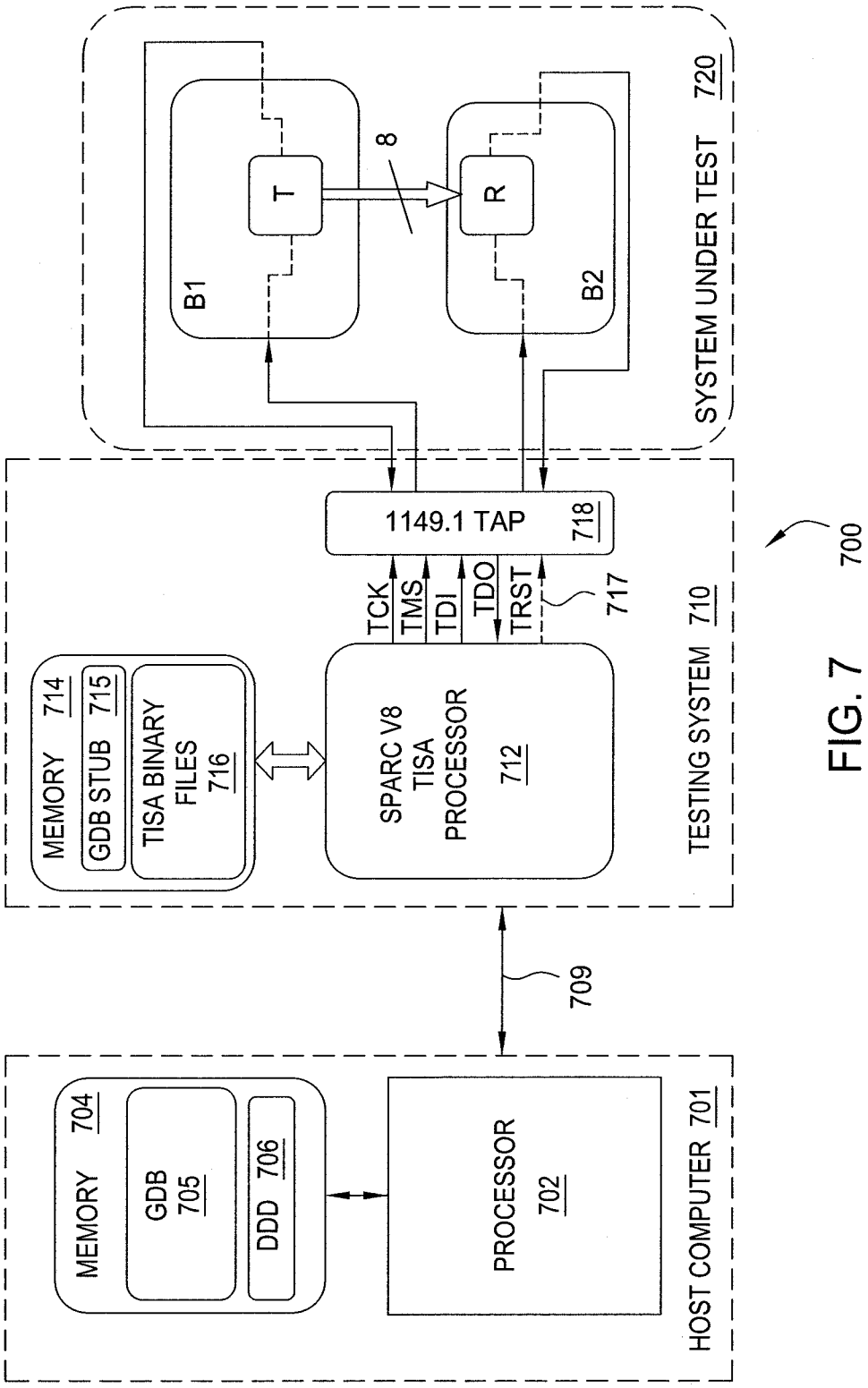
SYSTEM UNDER TEST 720
B1
T
8
R
B2
1149.1 TAP 718
TCK
TMS
TDI
TDO
TRST
717
MEMORY 714
GDB STUB 715
TISA BINARY FILES 716
SPARC V8 TISA PROCESSOR 712
TESTING SYSTEM 710
700
709
MEMORY 704
GDB 705
DDD 706
PROCESSOR 702
HOST COMPUTER 701

FIG. 7

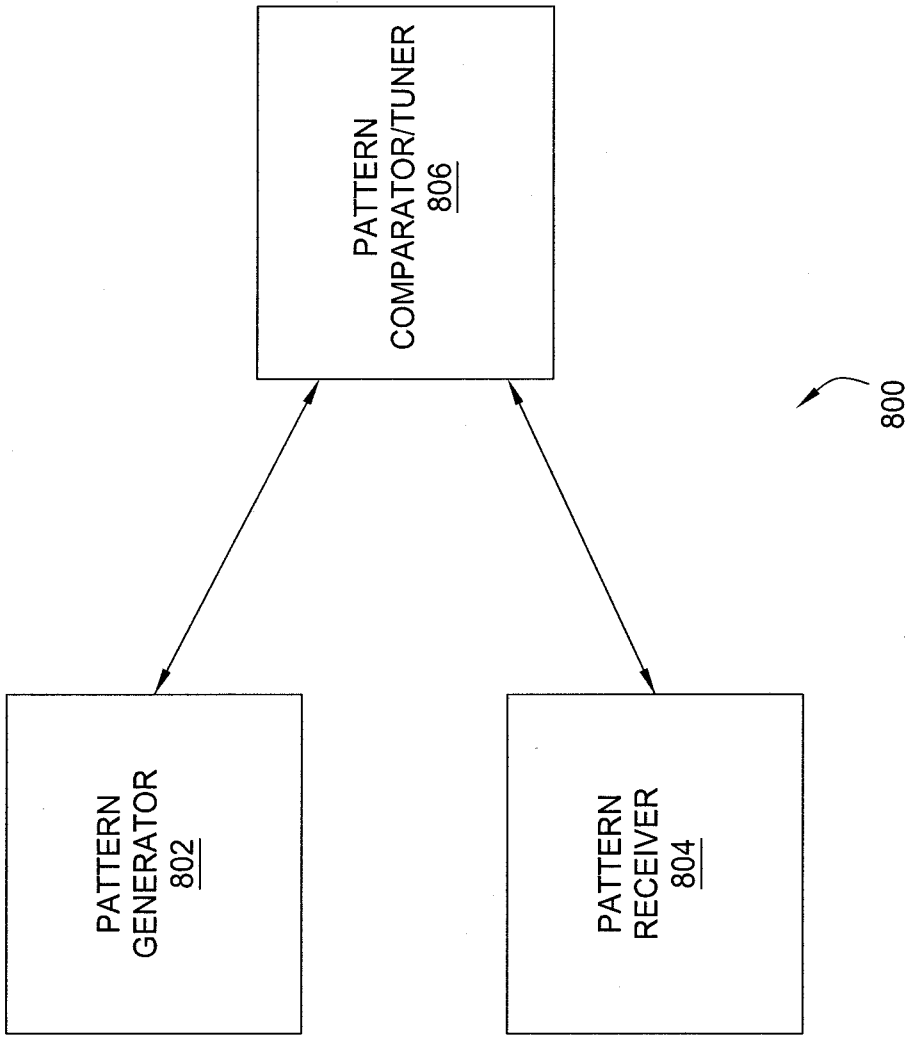
PATTERN COMPARATOR/TUNER 806
800
PATTERN GENERATOR 802
PATTERN RECEIVER 804

FIG. 8

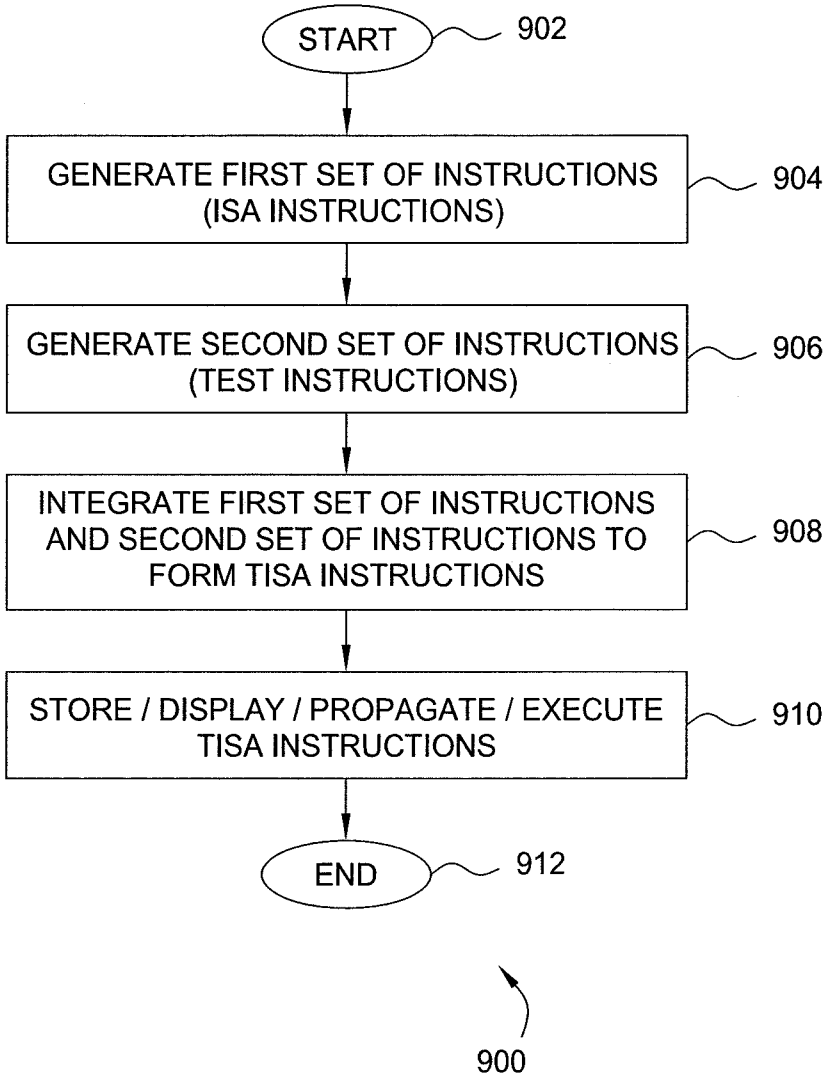
START
902
GENERATE FIRST SET OF INSTRUCTIONS (ISA INSTRUCTIONS)
904
GENERATE SECOND SET OF INSTRUCTIONS (TEST INSTRUCTIONS)
906
INTEGRATE FIRST SET OF INSTRUCTIONS AND SECOND SET OF INSTRUCTIONS TO FORM TISA INSTRUCTIONS
908
STORE / DISPLAY / PROPAGATE / EXECUTE TISA INSTRUCTIONS
910
END
912
900

FIG. 9

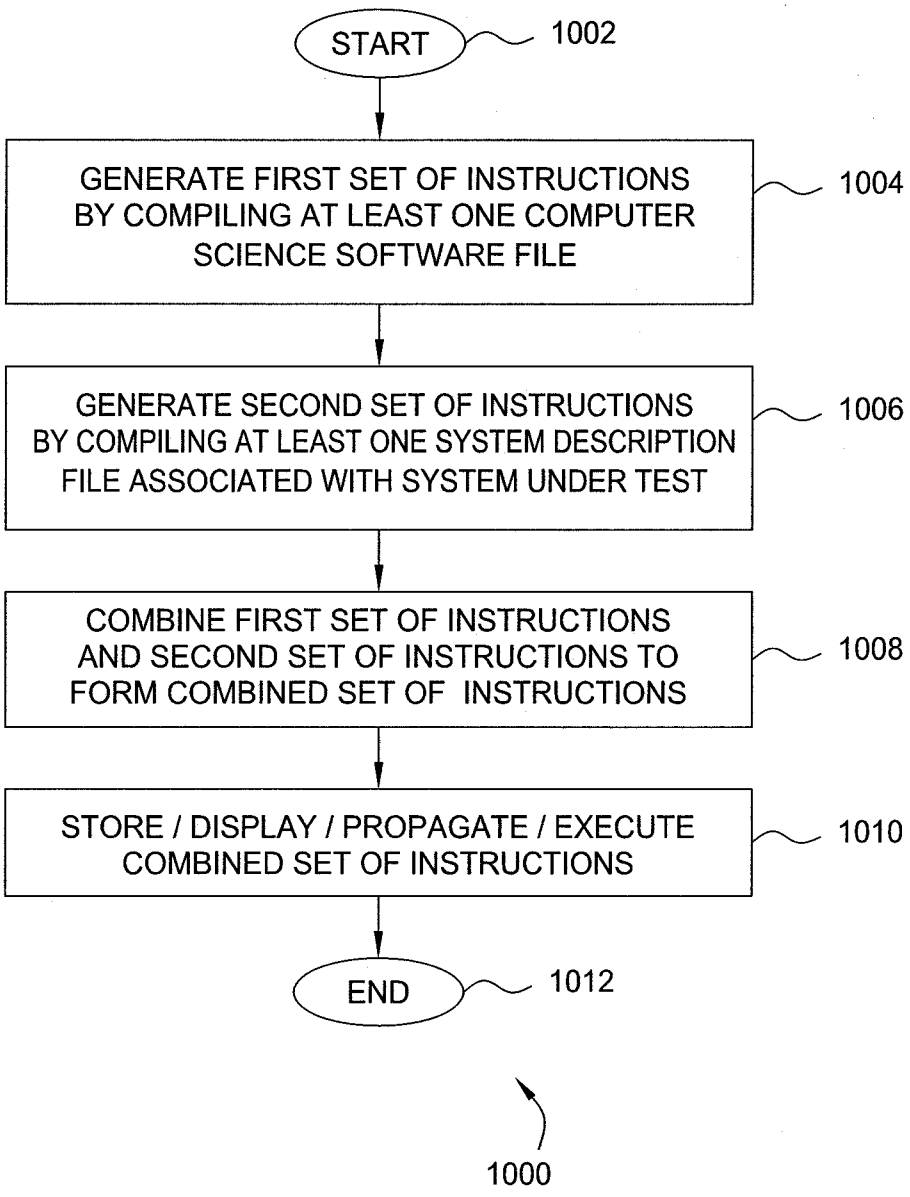
START
1002
GENERATE FIRST SET OF INSTRUCTIONS BY COMPILING AT LEAST ONE COMPUTER SCIENCE SOFTWARE FILE
1004
GENERATE SECOND SET OF INSTRUCTIONS BY COMPILING AT LEAST ONE SYSTEM DESCRIPTION FILE ASSOCIATED WITH SYSTEM UNDER TEST
1006
COMBINE FIRST SET OF INSTRUCTIONS AND SECOND SET OF INSTRUCTIONS TO FORM COMBINED SET OF INSTRUCTIONS
1008
STORE / DISPLAY / PROPAGATE / EXECUTE COMBINED SET OF INSTRUCTIONS
1010
END
1012
1000

FIG. 10

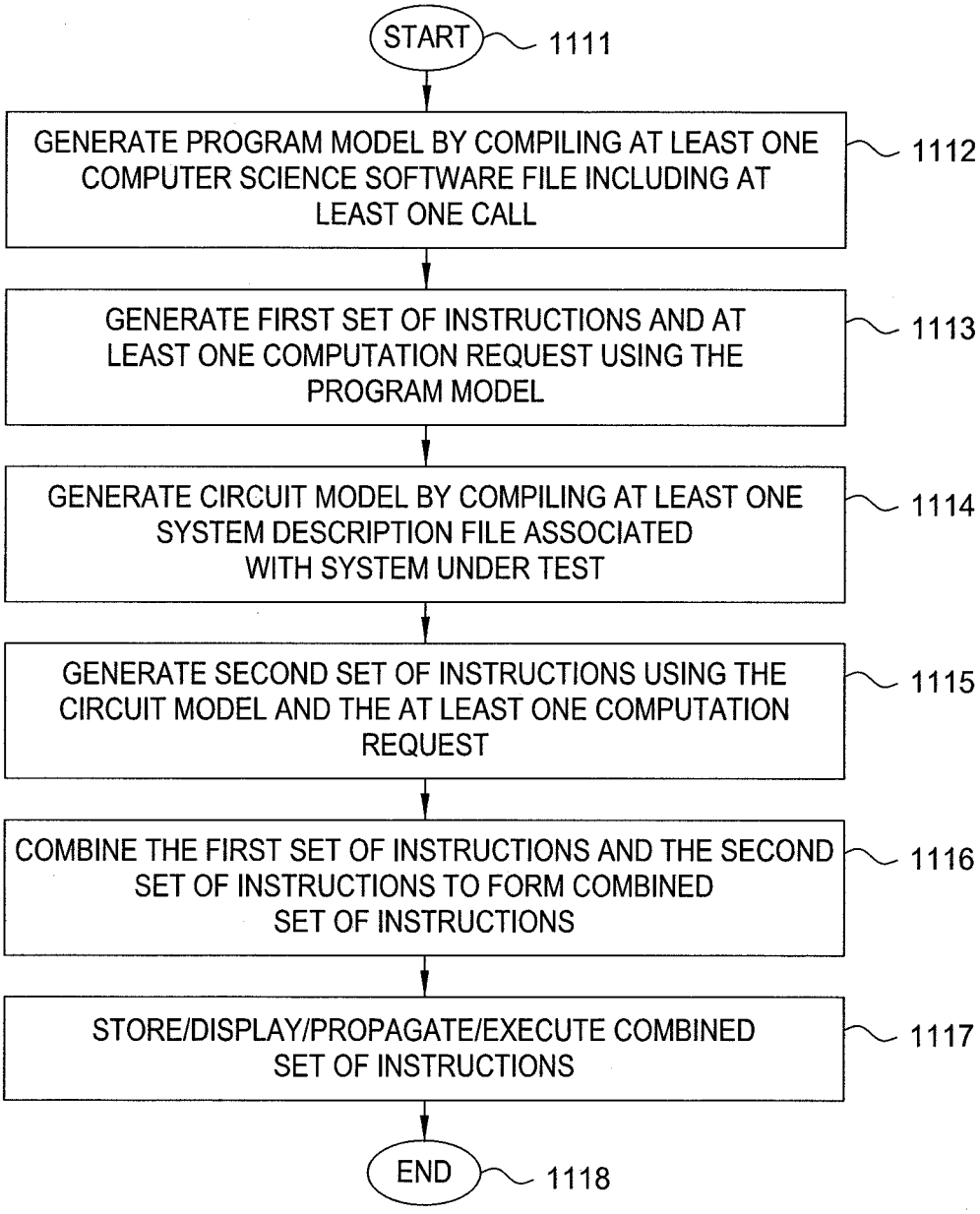
START
1111
GENERATE PROGRAM MODEL BY COMPILING AT LEAST ONE COMPUTER SCIENCE SOFTWARE FILE INCLUDING AT LEAST ONE CALL
1112
GENERATE FIRST SET OF INSTRUCTIONS AND AT LEAST ONE COMPUTATION REQUEST USING THE PROGRAM MODEL
1113
GENERATE CIRCUIT MODEL BY COMPILING AT LEAST ONE SYSTEM DESCRIPTION FILE ASSOCIATED WITH SYSTEM UNDER TEST
1114
GENERATE SECOND SET OF INSTRUCTIONS USING THE CIRCUIT MODEL AND THE AT LEAST ONE COMPUTATION REQUEST
1115
COMBINE THE FIRST SET OF INSTRUCTIONS AND THE SECOND SET OF INSTRUCTIONS TO FORM COMBINED SET OF INSTRUCTIONS
1116
STORE/DISPLAY/PROPAGATE/EXECUTE COMBINED SET OF INSTRUCTIONS
1117
END
1118

FIG. 11A

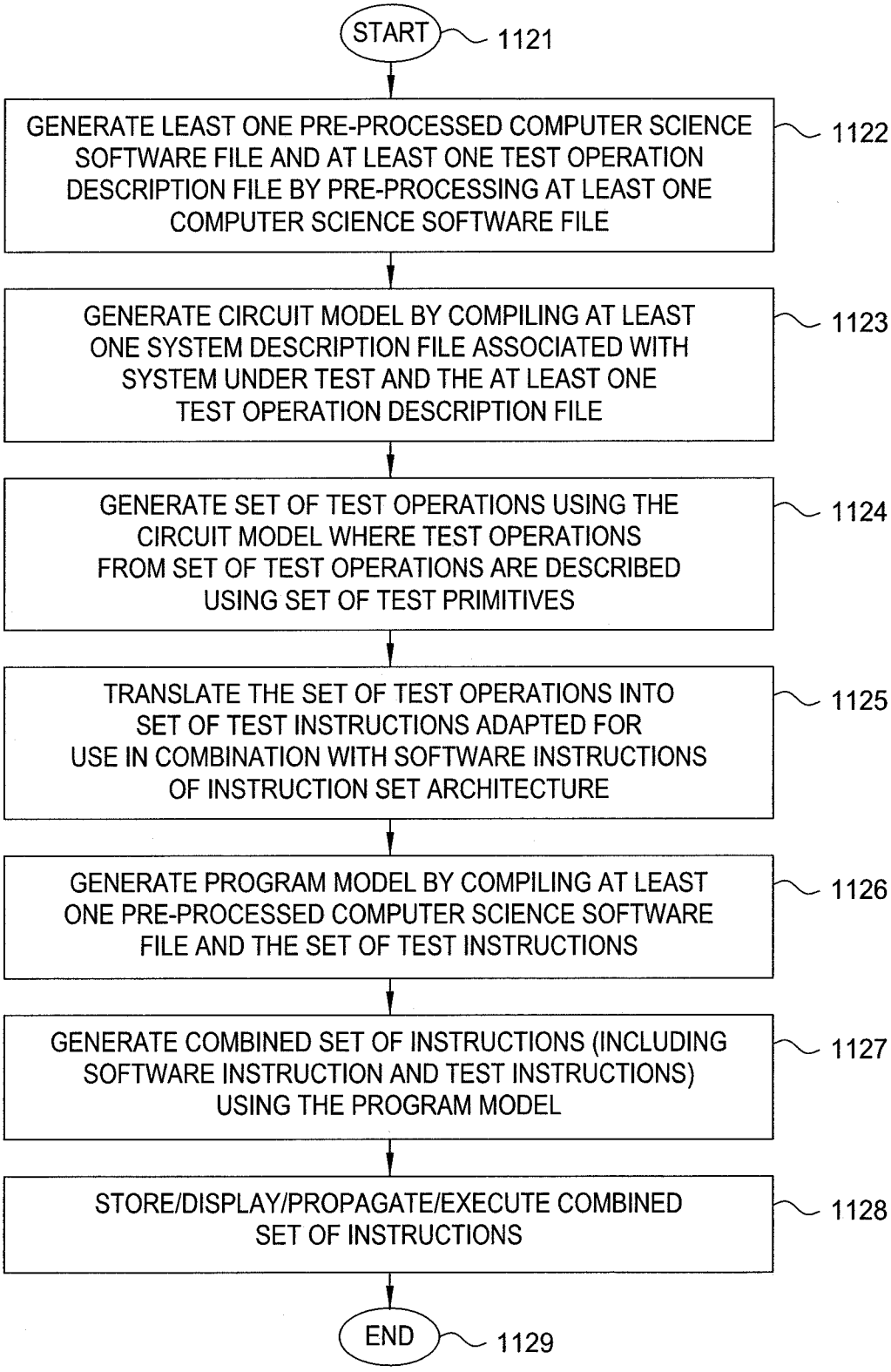
START
1121
GENERATE LEAST ONE PRE-PROCESSED COMPUTER SCIENCE SOFTWARE FILE AND AT LEAST ONE TEST OPERATION DESCRIPTION FILE BY PRE-PROCESSING AT LEAST ONE COMPUTER SCIENCE SOFTWARE FILE
1122
GENERATE CIRCUIT MODEL BY COMPILING AT LEAST ONE SYSTEM DESCRIPTION FILE ASSOCIATED WITH SYSTEM UNDER TEST AND THE AT LEAST ONE TEST OPERATION DESCRIPTION FILE
1123
GENERATE SET OF TEST OPERATIONS USING THE CIRCUIT MODEL WHERE TEST OPERATIONS FROM SET OF TEST OPERATIONS ARE DESCRIBED USING SET OF TEST PRIMITIVES
1124
TRANSLATE THE SET OF TEST OPERATIONS INTO SET OF TEST INSTRUCTIONS ADAPTED FOR USE IN COMBINATION WITH SOFTWARE INSTRUCTIONS OF INSTRUCTION SET ARCHITECTURE
1125
GENERATE PROGRAM MODEL BY COMPILING AT LEAST ONE PRE-PROCESSED COMPUTER SCIENCE SOFTWARE FILE AND THE SET OF TEST INSTRUCTIONS
1126
GENERATE COMBINED SET OF INSTRUCTIONS (INCLUDING SOFTWARE INSTRUCTION AND TEST INSTRUCTIONS) USING THE PROGRAM MODEL
1127
STORE/DISPLAY/PROPAGATE/EXECUTE COMBINED SET OF INSTRUCTIONS
1128
END
1129

FIG. 11B

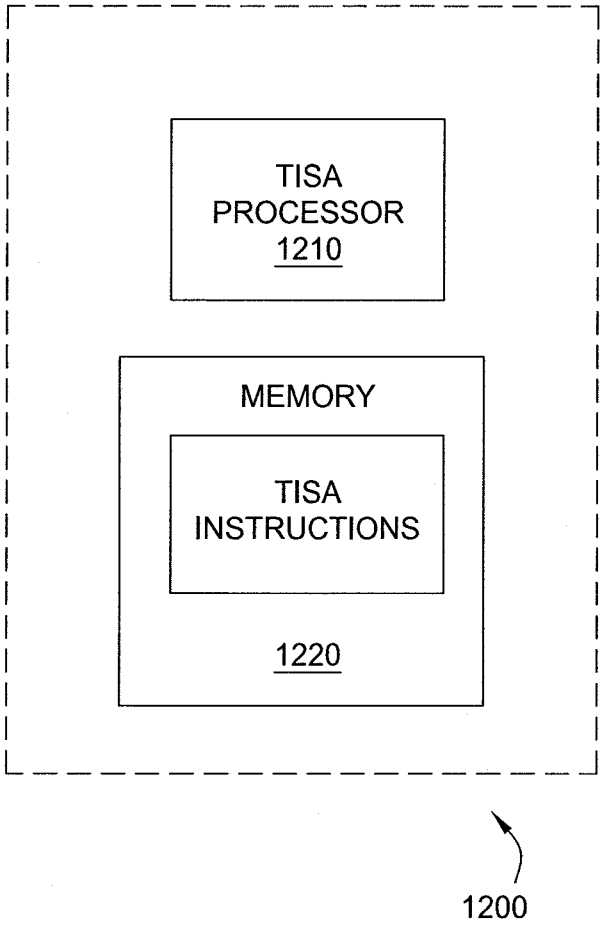

FIG. 12

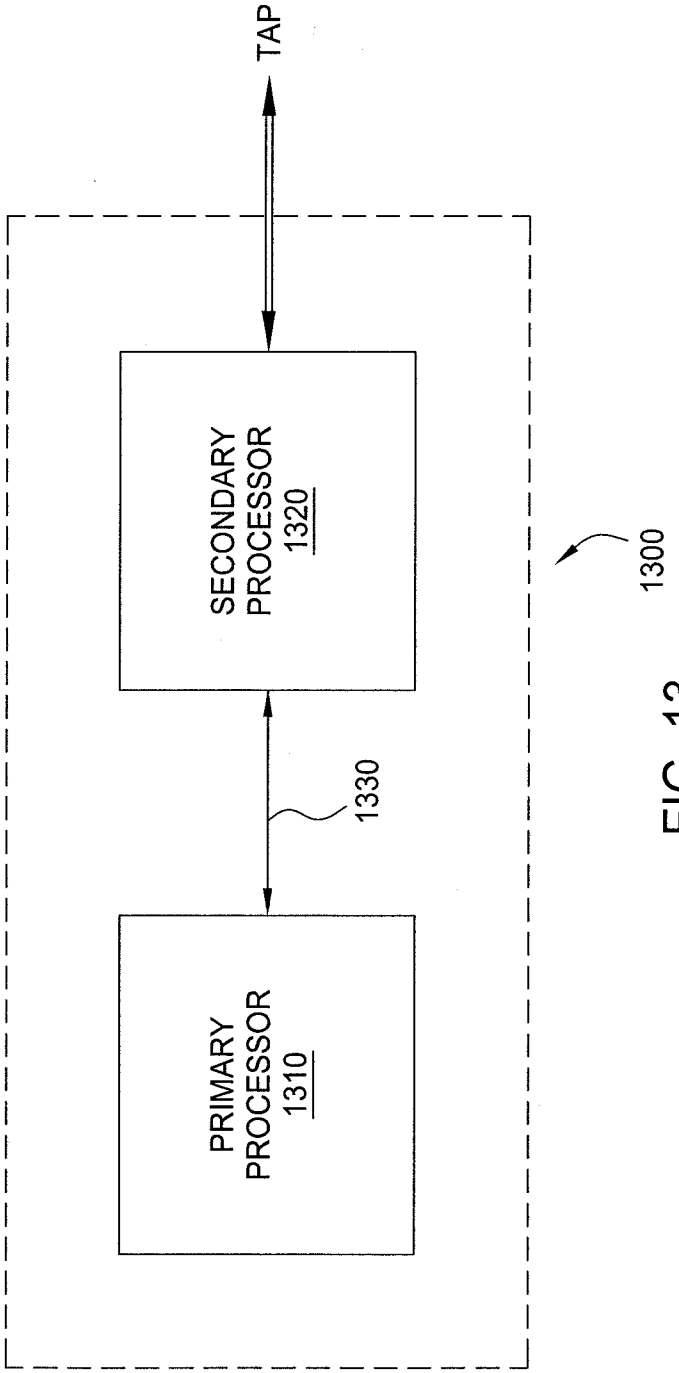
TAP
SECONDARY PROCESSOR 1320
1330
PRIMARY PROCESSOR 1310
1300

FIG. 13

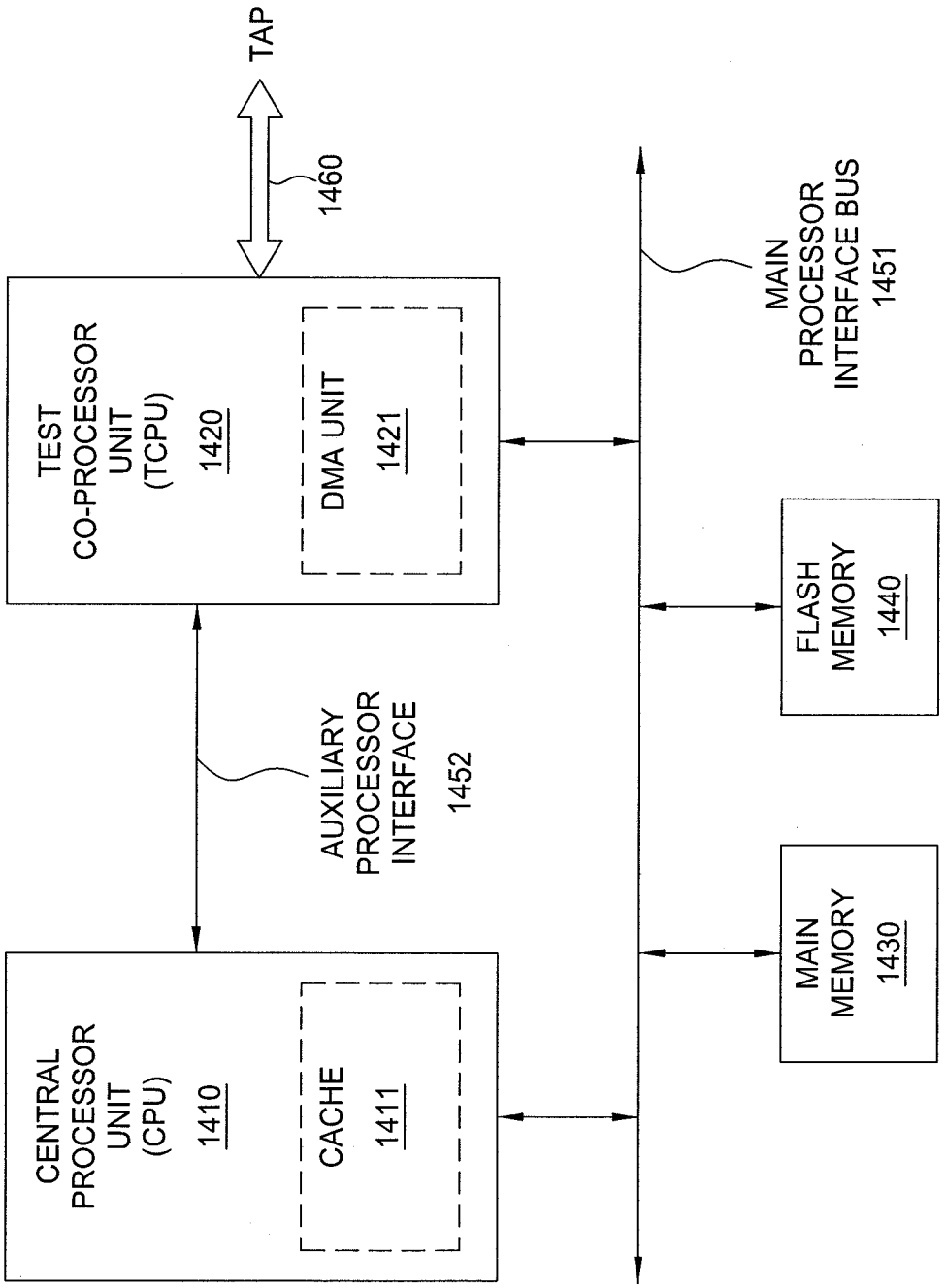
TAP
1460
TEST CO-PROCESSOR UNIT (TCPU) 1420
DMA UNIT 1421
MAIN PROCESSOR INTERFACE BUS 1451
FLASH MEMORY 1440
1400
AUXILIARY PROCESSOR INTERFACE 1452
MAIN MEMORY 1430
CENTRAL PROCESSOR UNIT (CPU) 1410
CACHE 1411

FIG. 14

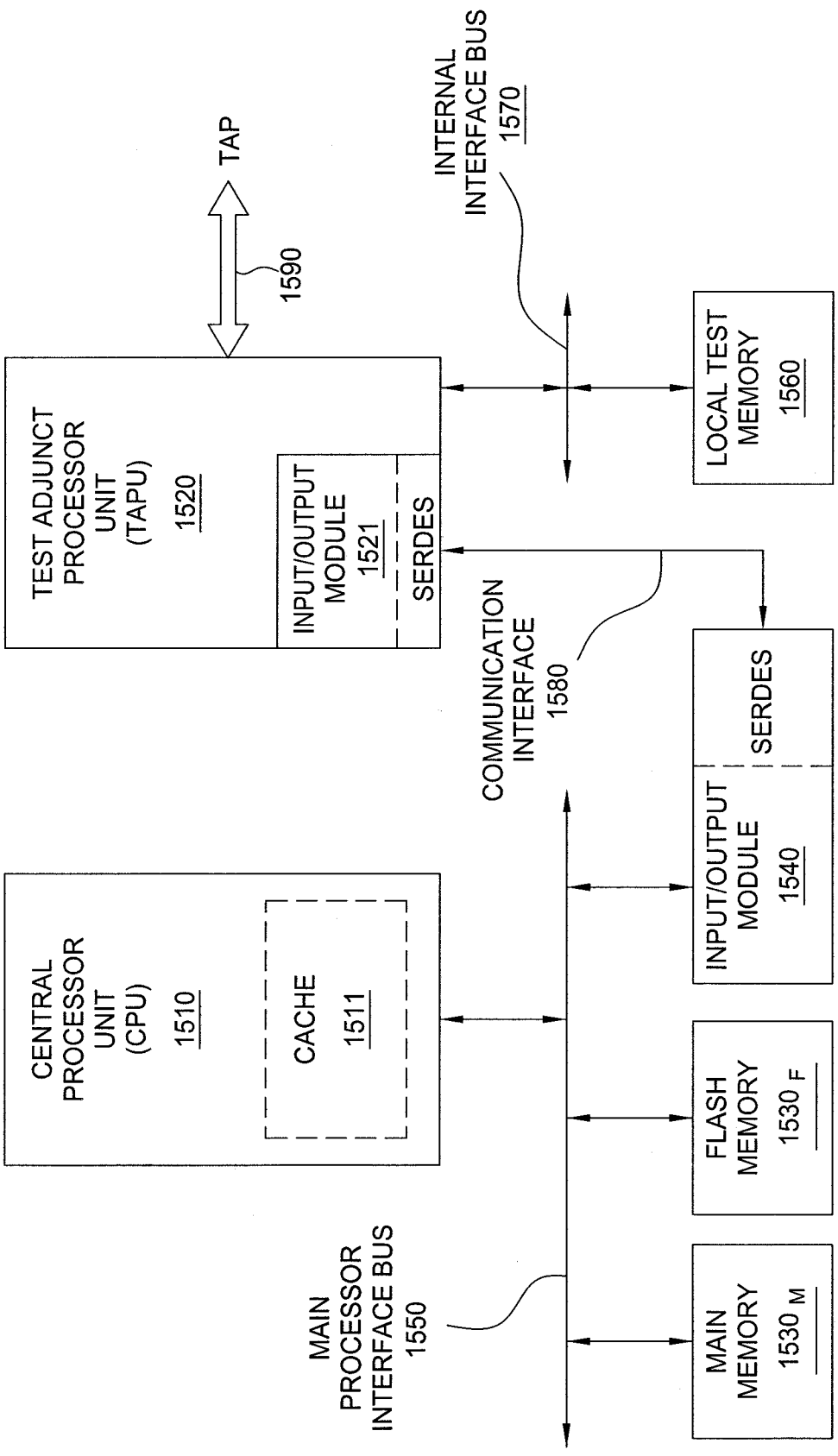
TAP
1590
INTERNAL INTERFACE BUS 1570
TEST ADJUNCT PROCESSOR UNIT (TAPU) 1520
INPUT/OUTPUT MODULE 1521
SERDES
LOCAL TEST MEMORY 1560
COMMUNICATION INTERFACE 1580
SERDES
INPUT/OUTPUT MODULE 1540
1500
CENTRAL PROCESSOR UNIT (CPU) 1510
CACHE 1511
FLASH MEMORY 1530 F
MAIN PROCESSOR INTERFACE BUS 1550
MAIN MEMORY 1530 M

FIG. 15

StatusRegister
ControlRegister
BlockRegister
ScanLengthRegister
ScanStateRegister
UserDataRegister1
UserDataRegister2
• • •
UserDataRegister11

R01 Test Processor User Accessible Data Registers

FIG. 16A

BlockPointerRegister
BlockCountRegister
InstructionRegister

R02 Internal Scratch Registers

FIG. 16B

BlockOffsetField
ScanLengthField
StateTraversalField
SourceLocationField
DestinationLocationField
ExpectedValueField
ResponseLocationField
MaskField
ResultLocationField

R03 Scatter/Gather Segment Descriptions

FIG. 16C

BlockOffsetField
BlockCountField
ScatterGatherOpcodeField1
ScatterGatherBlockField1
• • •
ScatterGatherOpcodeFieldN
ScatterGatherBlockFieldN

R04 MultiBlock Scatter/Gather Segment Descriptions

FIG. 16D

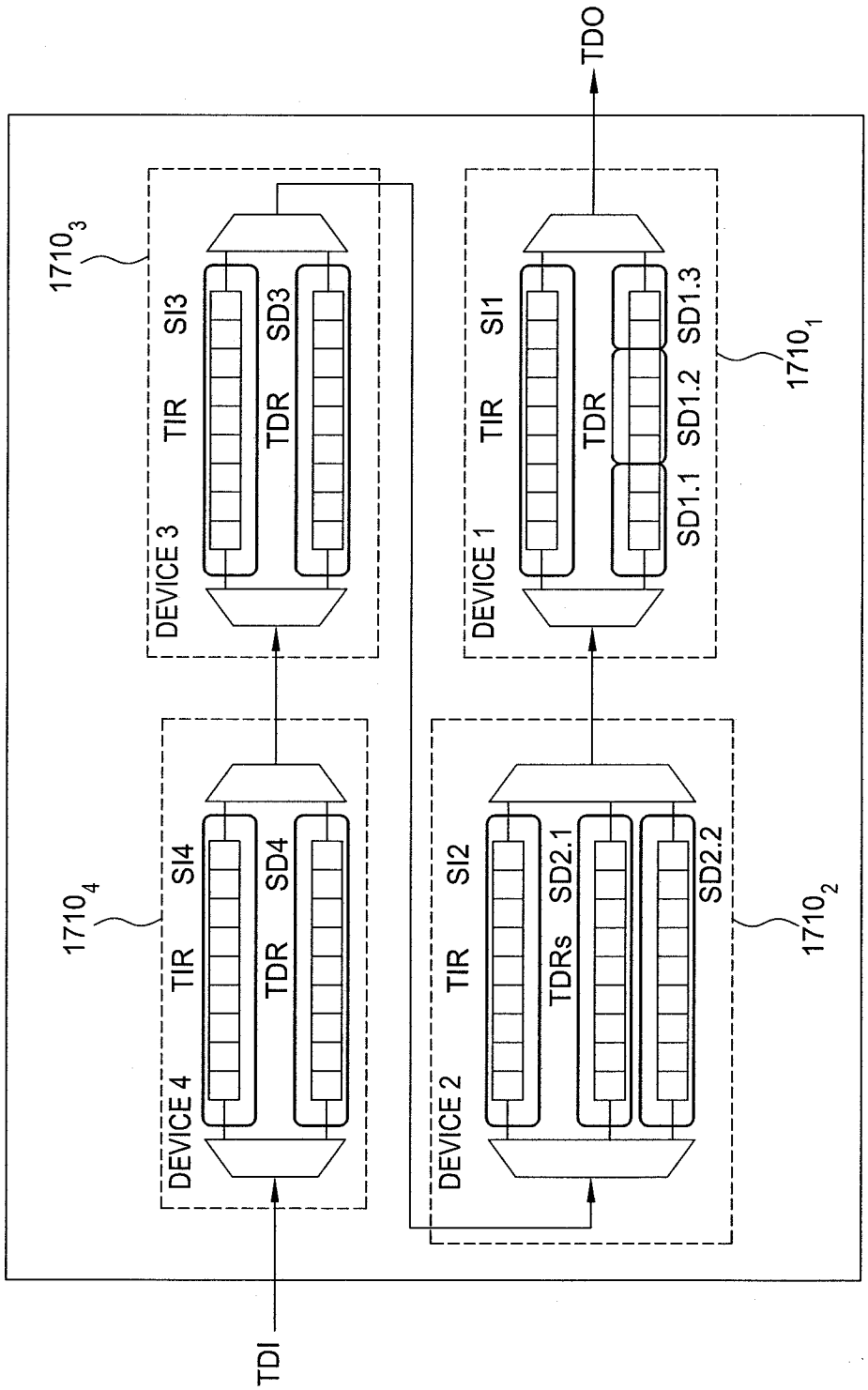
TDO
TDI
1700
1710$_3$
DEVICE 3
TIR
SI3
TDR
SD3
1710$_1$
DEVICE 1
TIR
SI1
TDR
SD1.1
SD1.2
SD1.3
1710$_4$
DEVICE 4
TIR
SI4
TDR
SD4
1710$_2$
DEVICE 2
TIR
SI2
TDRs
SD2.1
SD2.2

FIG. 17

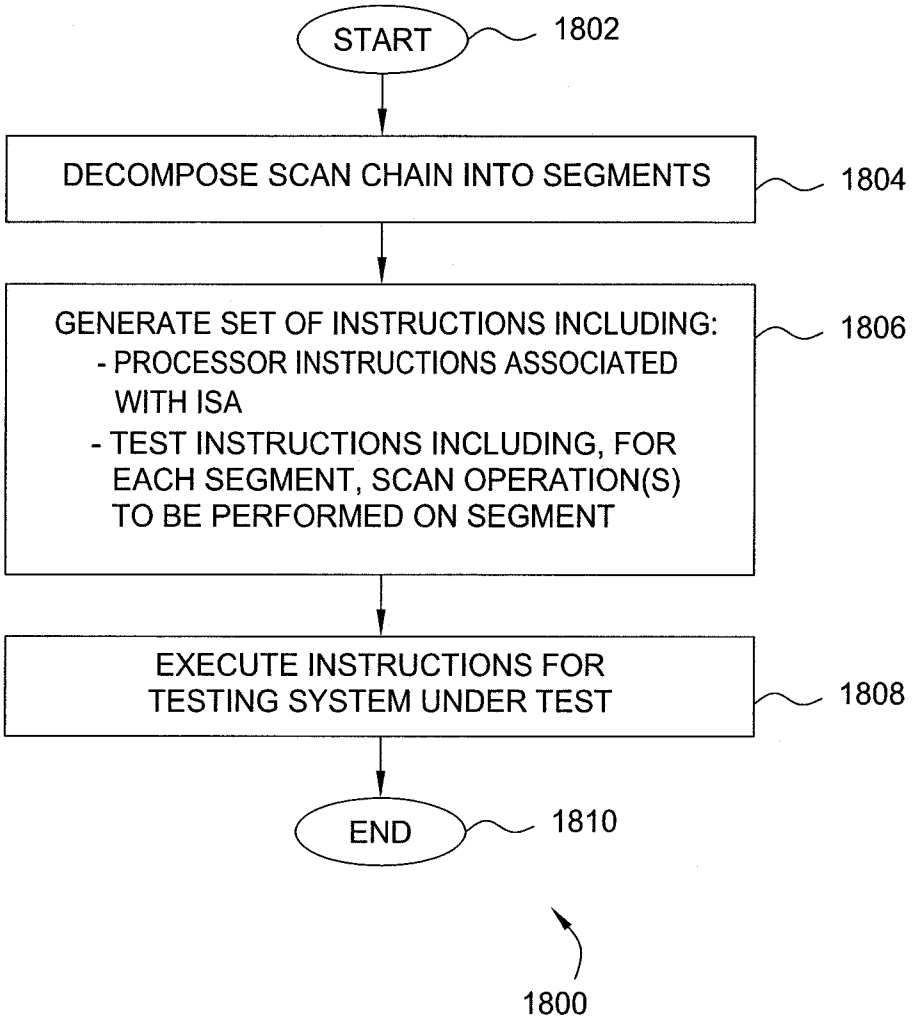
START
1802
DECOMPOSE SCAN CHAIN INTO SEGMENTS
1804
GENERATE SET OF INSTRUCTIONS INCLUDING:
- PROCESSOR INSTRUCTIONS ASSOCIATED WITH ISA
- TEST INSTRUCTIONS INCLUDING, FOR EACH SEGMENT, SCAN OPERATION(S) TO BE PERFORMED ON SEGMENT
1806
EXECUTE INSTRUCTIONS FOR TESTING SYSTEM UNDER TEST
1808
END
1810
1800

FIG. 18

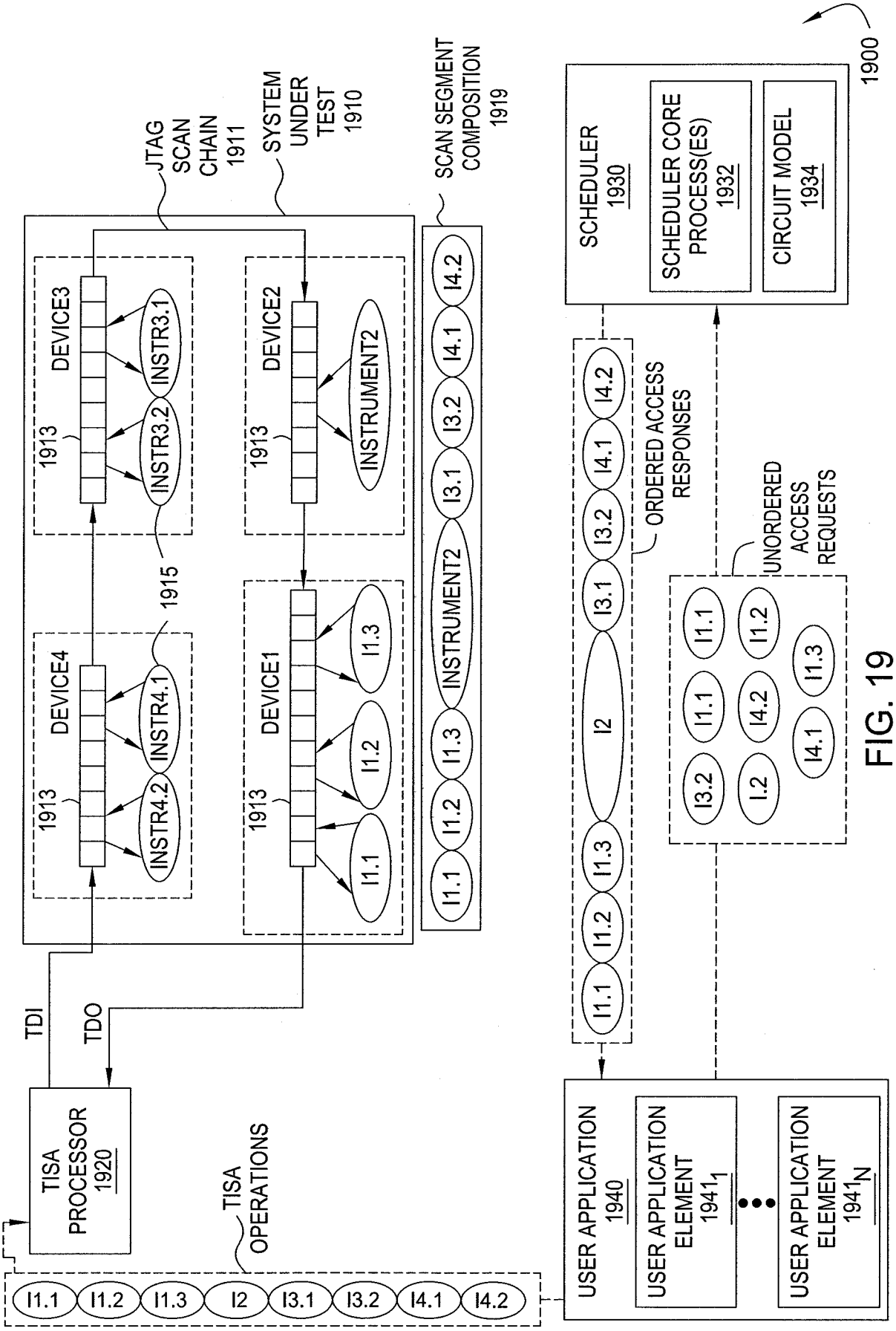
1900
SCHEDULER 1930
SCHEDULER CORE PROCESS(ES) 1932
CIRCUIT MODEL 1934
ORDERED ACCESS RESPONSES
UNORDERED ACCESS REQUESTS
I1.1 I1.2 I1.3 I2 I3.1 I3.2 I4.1 I4.2
USER APPLICATION 1940
USER APPLICATION ELEMENT 1941$_1$
USER APPLICATION ELEMENT 1941$_N$
TISA OPERATIONS
TISA PROCESSOR 1920
TDI
TDO
JTAG SCAN CHAIN 1911
SYSTEM UNDER TEST 1910
SCAN SEGMENT COMPOSITION 1919
DEVICE3
DEVICE4
DEVICE2
DEVICE1
1913
1915
INSTR3.1
INSTR3.2
INSTR4.1
INSTR4.2
INSTRUMENT2

FIG. 19

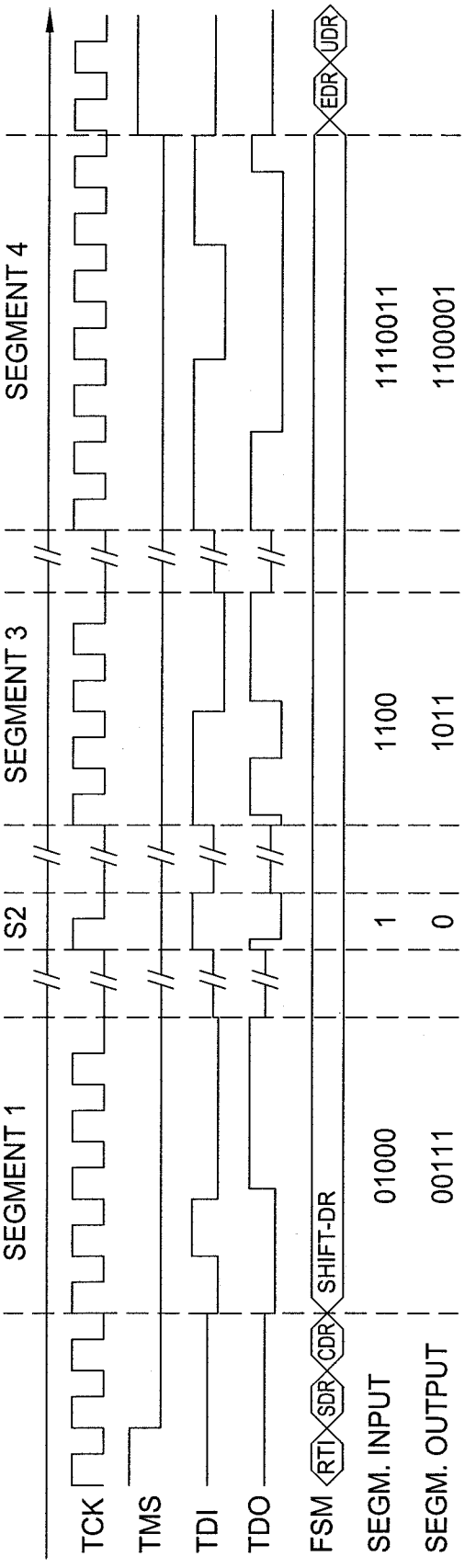
SEGMENT 1
S2
SEGMENT 3
SEGMENT 4
TCK
TMS
TDI
TDO
FSM
SEGM. INPUT
SEGM. OUTPUT
RTI
SDR
CDR
SHIFT-DR
EDR
UDR
01000
00111
1
0
1100
1011
1110011
1100001
2000

FIG. 20

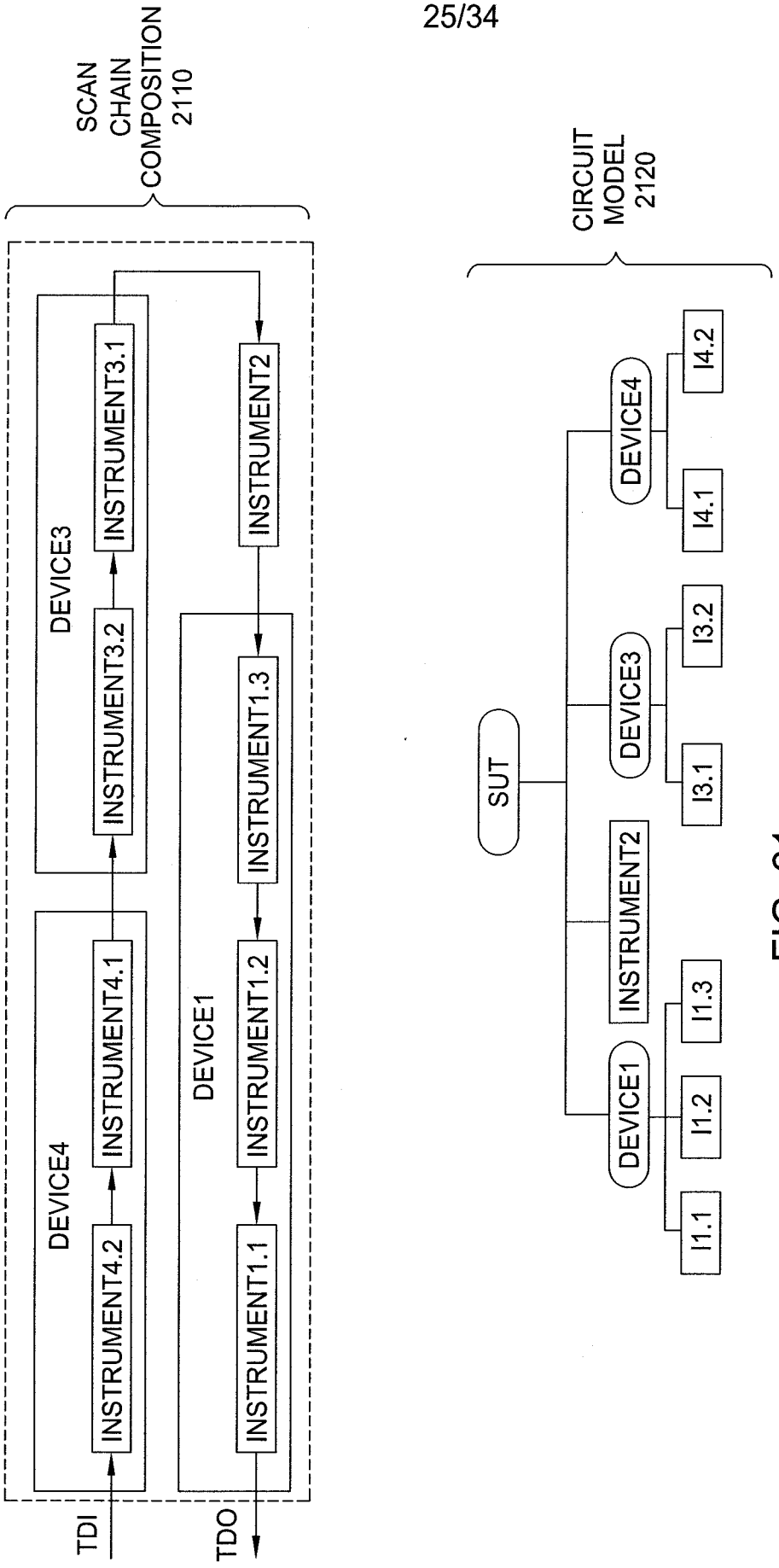
SCAN CHAIN COMPOSITION 2110
DEVICE3
DEVICE4
DEVICE1
INSTRUMENT3.1
INSTRUMENT3.2
INSTRUMENT4.1
INSTRUMENT4.2
INSTRUMENT2
INSTRUMENT1.3
INSTRUMENT1.2
INSTRUMENT1.1
TDI
TDO
CIRCUIT MODEL 2120
SUT
DEVICE4
DEVICE3
INSTRUMENT2
DEVICE1
I4.2
I4.1
I3.2
I3.1
I1.3
I1.2
I1.1

FIG. 21

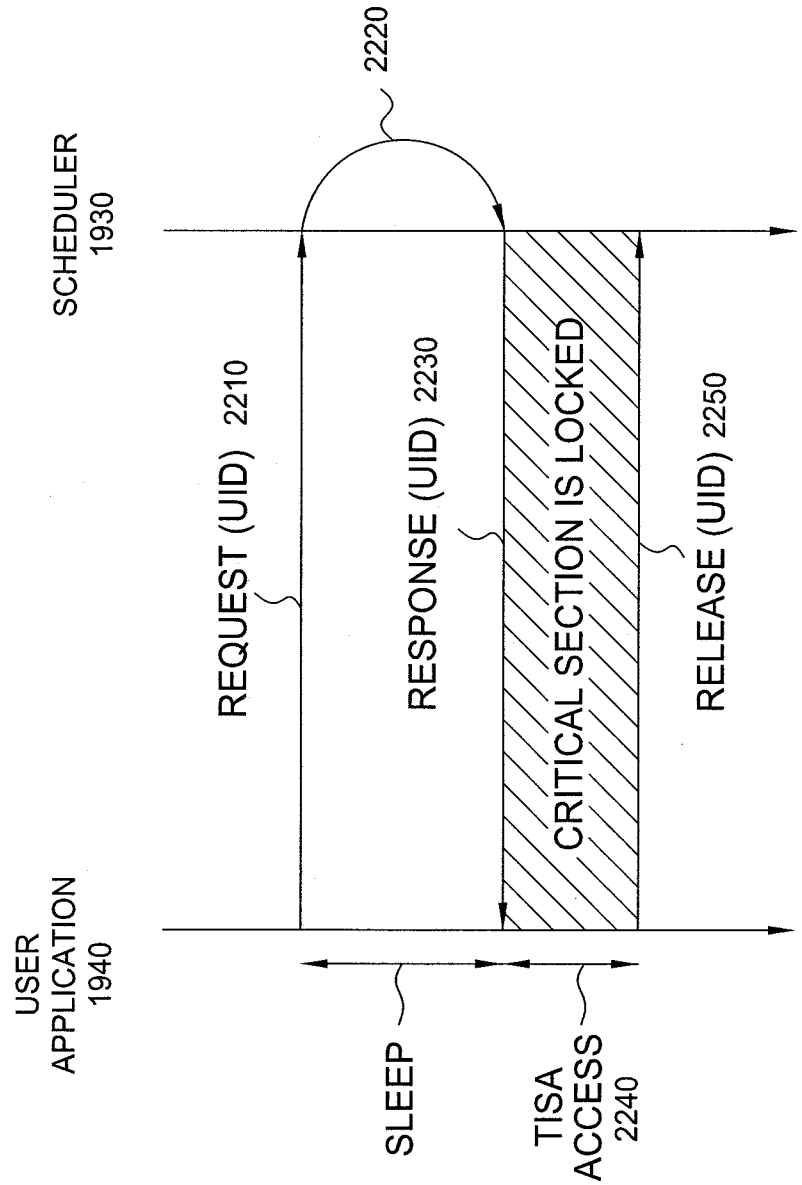
2220
SCHEDULER
1930
REQUEST (UID) 2210
RESPONSE (UID) 2230
CRITICAL SECTION IS LOCKED
RELEASE (UID) 2250
2200
USER
APPLICATION
1940
SLEEP
TISA
ACCESS
2240

FIG. 22

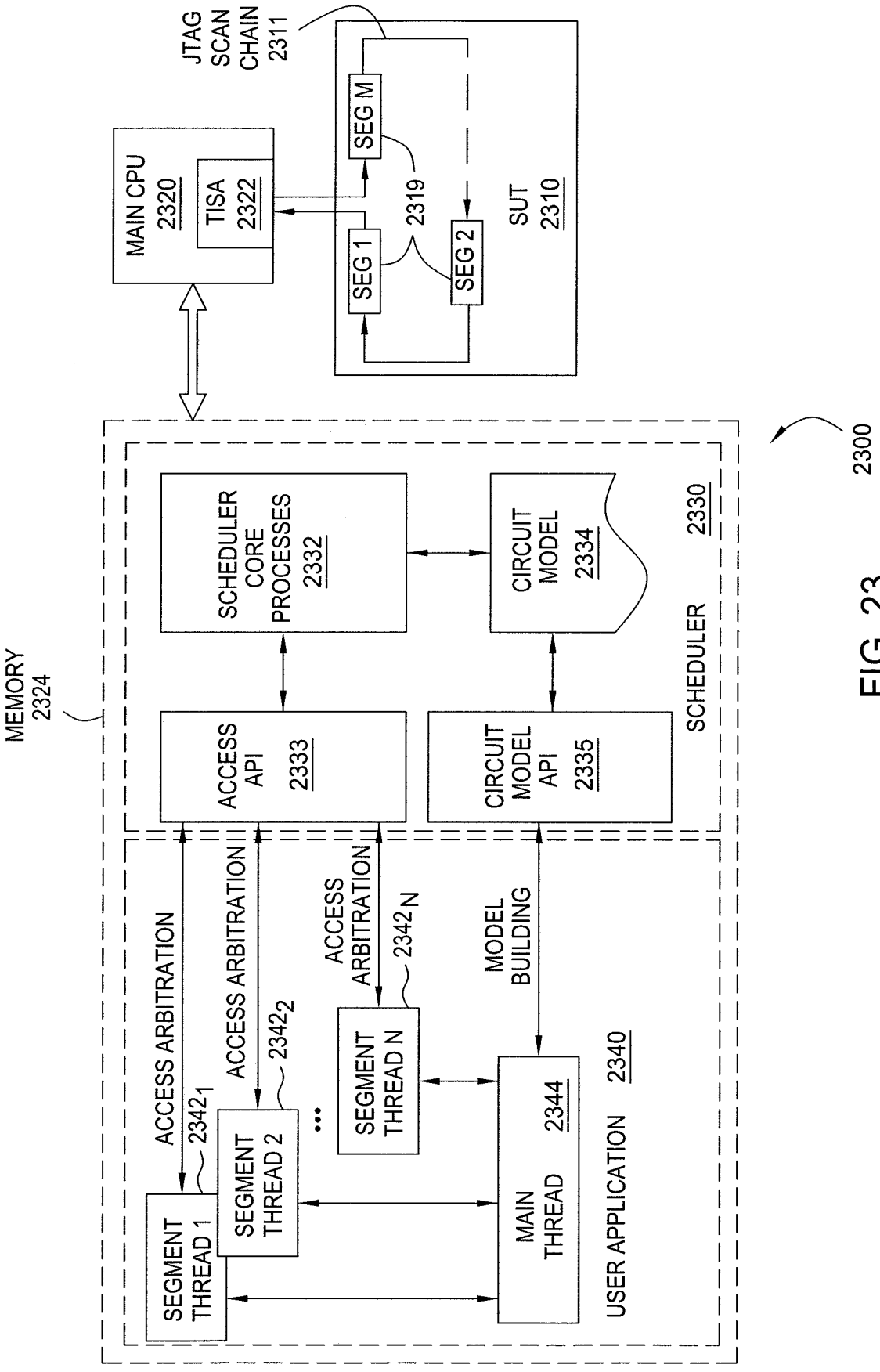
JTAG SCAN CHAIN 2311
MAIN CPU 2320
TISA 2322
SEG M
SEG 1
SEG 2
2319
SUT 2310
2300
MEMORY 2324
SCHEDULER CORE PROCESSES 2332
CIRCUIT MODEL 2334
SCHEDULER 2330
ACCESS API 2333
CIRCUIT MODEL API 2335
ACCESS ARBITRATION
ACCESS ARBITRATION
ACCESS ARBITRATION
MODEL BUILDING
2342 N
2342 2
2342 1
SEGMENT THREAD 1
SEGMENT THREAD 2
SEGMENT THREAD N
MAIN THREAD 2344
USER APPLICATION 2340

FIG. 23

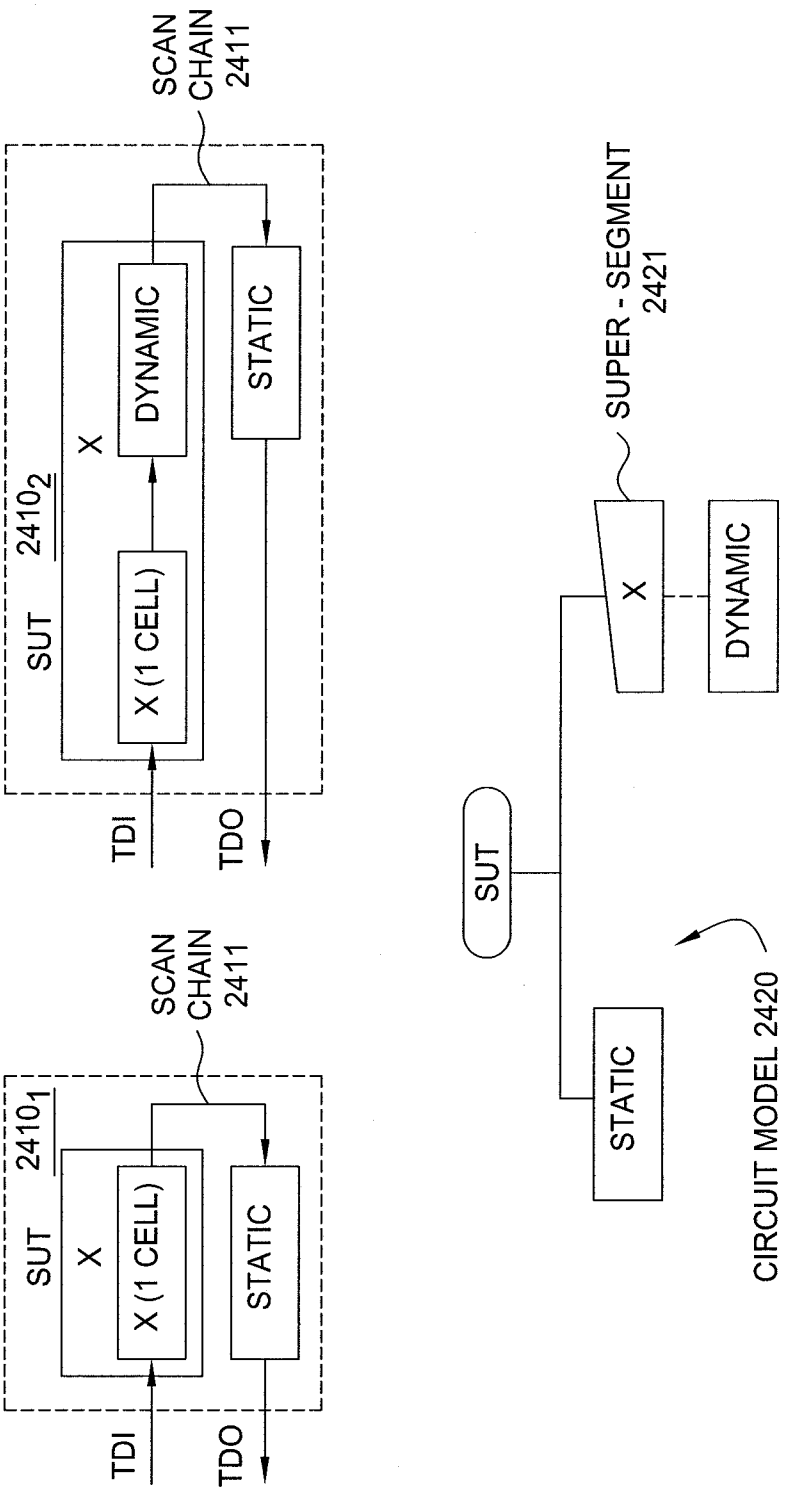
SCAN CHAIN 2411
SUT 2410 2
X
DYNAMIC
X (1 CELL)
STATIC
TDI
TDO
SCAN CHAIN 2411
SUT 2410 1
X
X (1 CELL)
STATIC
TDI
TDO
SUPER - SEGMENT 2421
SUT
X
DYNAMIC
STATIC
CIRCUIT MODEL 2420

FIG. 24

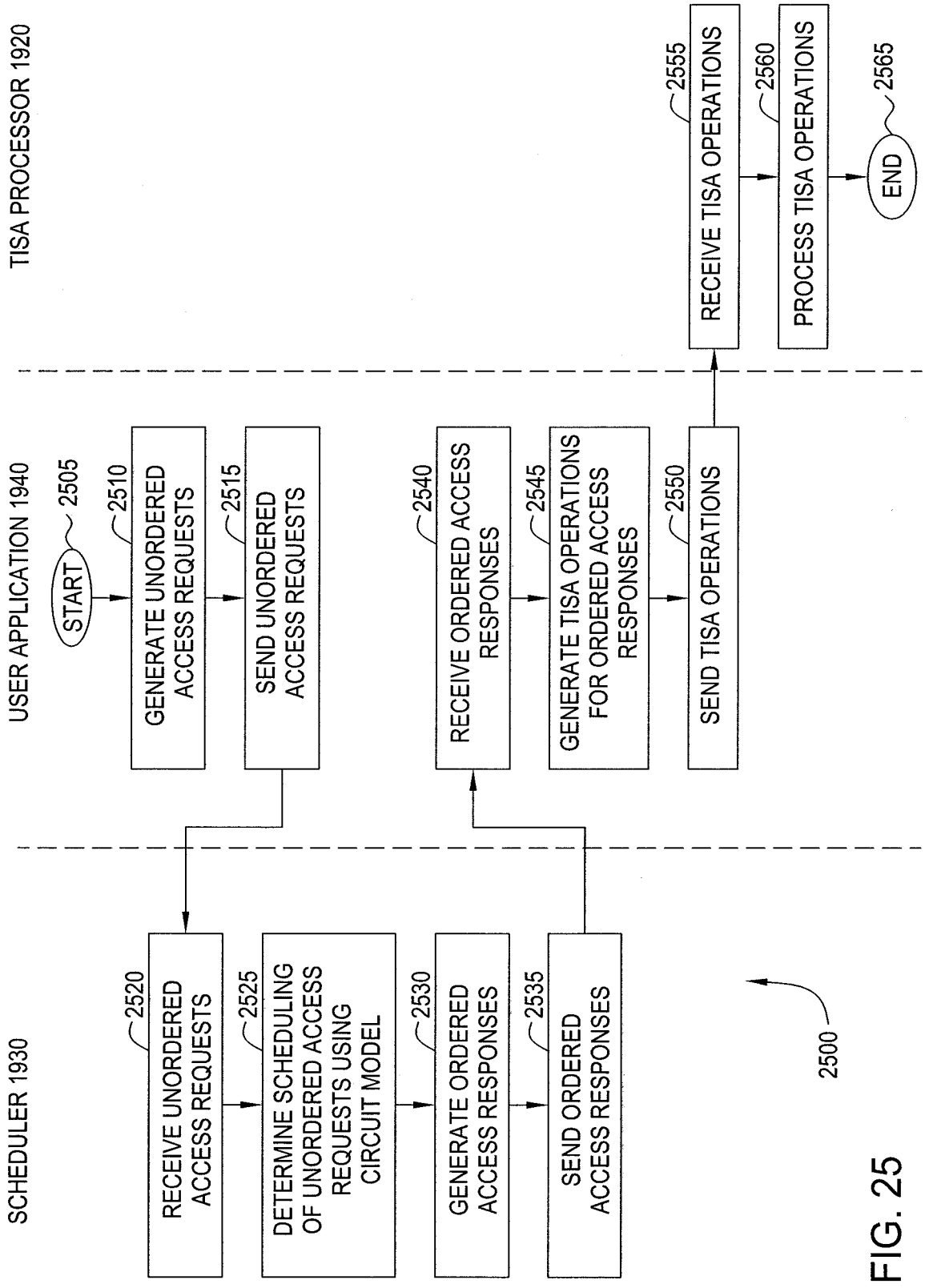
TISA PROCESSOR 1920
USER APPLICATION 1940
SCHEDULER 1930
START
2505
GENERATE UNORDERED ACCESS REQUESTS
2510
SEND UNORDERED ACCESS REQUESTS
2515
RECEIVE UNORDERED ACCESS REQUESTS
2520
DETERMINE SCHEDULING OF UNORDERED ACCESS REQUESTS USING CIRCUIT MODEL
2525
GENERATE ORDERED ACCESS RESPONSES
2530
SEND ORDERED ACCESS RESPONSES
2535
RECEIVE ORDERED ACCESS RESPONSES
2540
GENERATE TISA OPERATIONS FOR ORDERED ACCESS RESPONSES
2545
SEND TISA OPERATIONS
2550
RECEIVE TISA OPERATIONS
2555
PROCESS TISA OPERATIONS
2560
END
2565
2500

FIG. 25

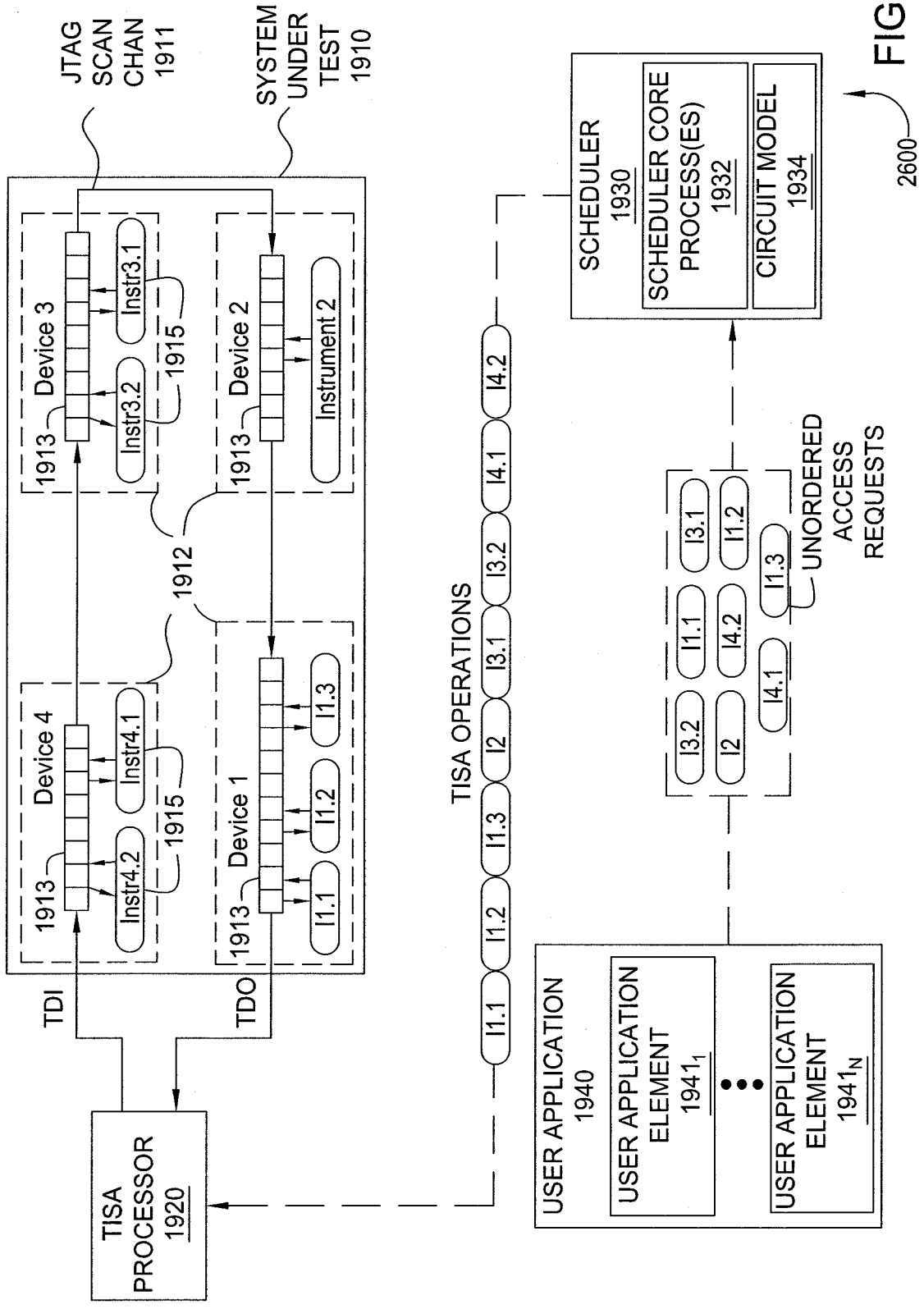
JTAG SCAN CHAN 1911
SYSTEM UNDER TEST 1910
Device 3
Device 4
Device 2
Device 1
1913
Instr3.1
Instr3.2
Instr4.1
Instr4.2
Instrument 2
1915
1912
I1.1
I1.2
I1.3
TDI
TDO
TISA PROCESSOR 1920
TISA OPERATIONS
I2
I3.1
I3.2
I4.1
I4.2
USER APPLICATION 1940
USER APPLICATION ELEMENT 1941$_1$
USER APPLICATION ELEMENT 1941$_N$
UNORDERED ACCESS REQUESTS
SCHEDULER 1930
SCHEDULER CORE PROCESS(ES) 1932
CIRCUIT MODEL 1934
2600

FIG. 26

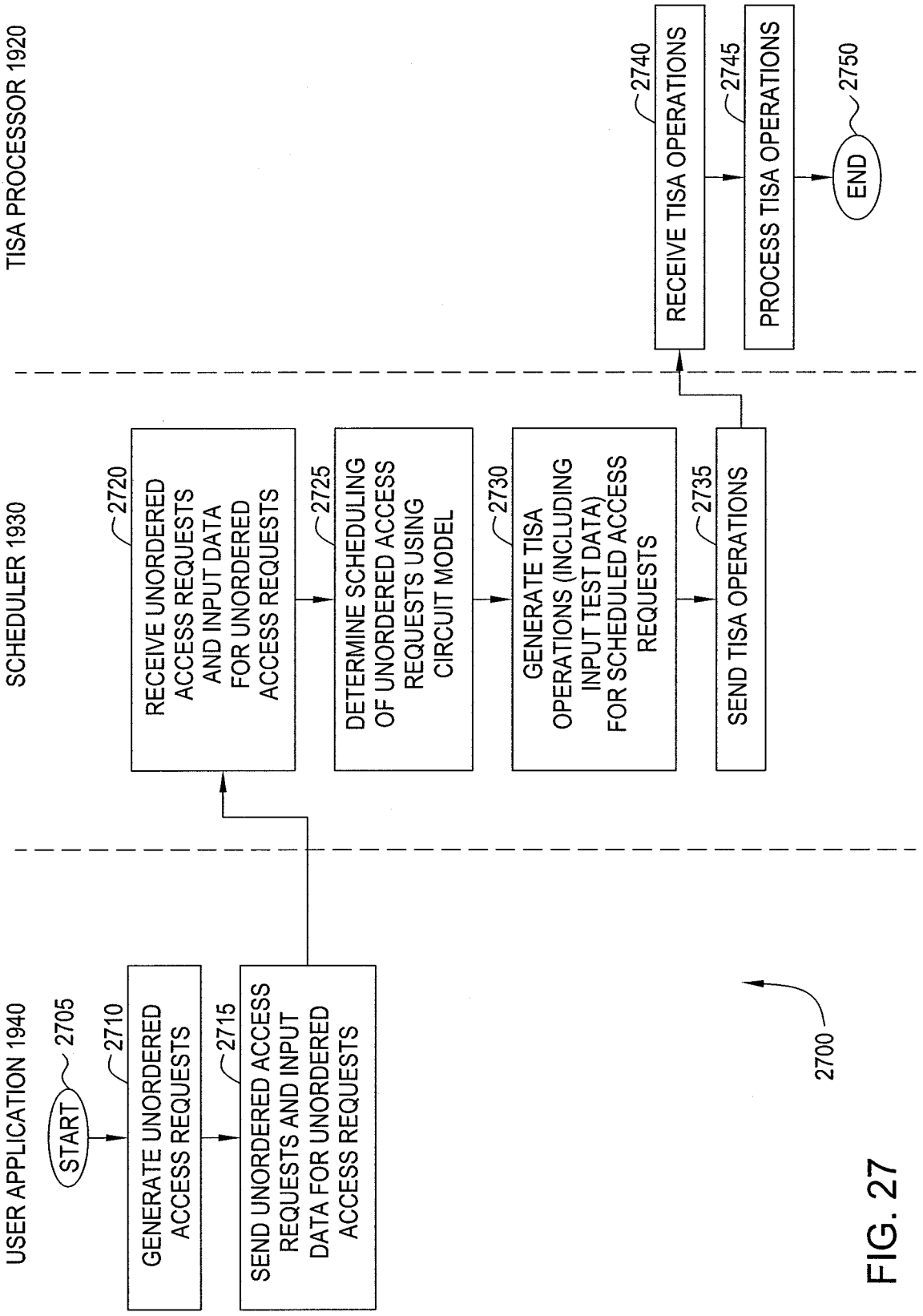
USER APPLICATION 1940
SCHEDULER 1930
TISA PROCESSOR 1920
START
2705
GENERATE UNORDERED ACCESS REQUESTS
2710
SEND UNORDERED ACCESS REQUESTS AND INPUT DATA FOR UNORDERED ACCESS REQUESTS
2715
RECEIVE UNORDERED ACCESS REQUESTS AND INPUT DATA FOR UNORDERED ACCESS REQUESTS
2720
DETERMINE SCHEDULING OF UNORDERED ACCESS REQUESTS USING CIRCUIT MODEL
2725
GENERATE TISA OPERATIONS (INCLUDING INPUT TEST DATA) FOR SCHEDULED ACCESS REQUESTS
2730
SEND TISA OPERATIONS
2735
RECEIVE TISA OPERATIONS
2740
PROCESS TISA OPERATIONS
2745
END
2750
2700

FIG. 27

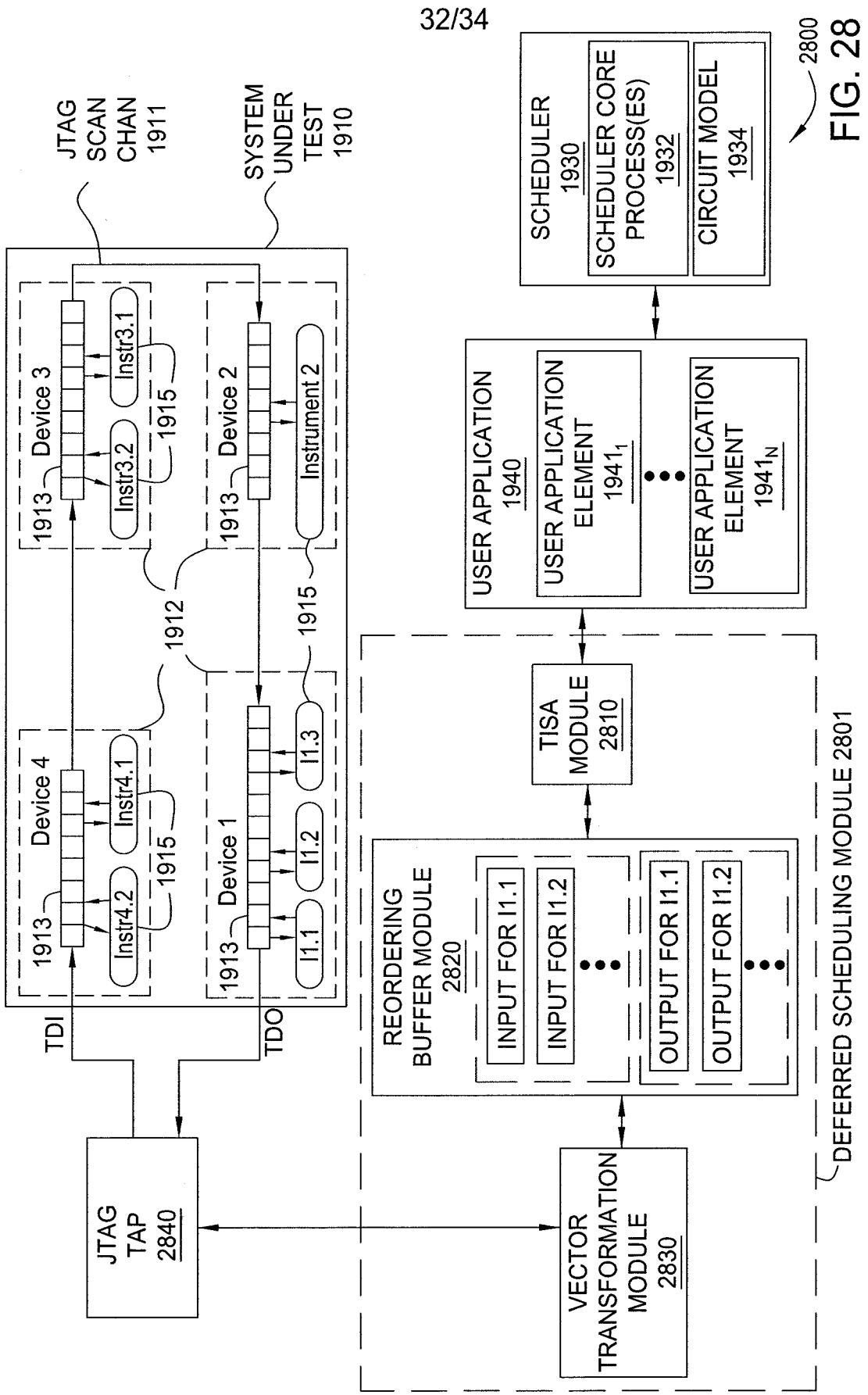
2800
SCHEDULER 1930
SCHEDULER CORE PROCESS(ES) 1932
CIRCUIT MODEL 1934
USER APPLICATION 1940
USER APPLICATION ELEMENT 1941$_1$
USER APPLICATION ELEMENT 1941$_N$
TISA MODULE 2810
REORDERING BUFFER MODULE 2820
INPUT FOR I1.1
INPUT FOR I1.2
OUTPUT FOR I1.1
OUTPUT FOR I1.2
VECTOR TRANSFORMATION MODULE 2830
JTAG TAP 2840
DEFERRED SCHEDULING MODULE 2801
JTAG SCAN CHAN 1911
SYSTEM UNDER TEST 1910
Device 4
Device 3
Device 2
Device 1
Instr4.1
Instr4.2
Instr3.1
Instr3.2
Instrument 2
I1.1
I1.2
I1.3
1913
1915
1912
TDI
TDO

FIG. 28

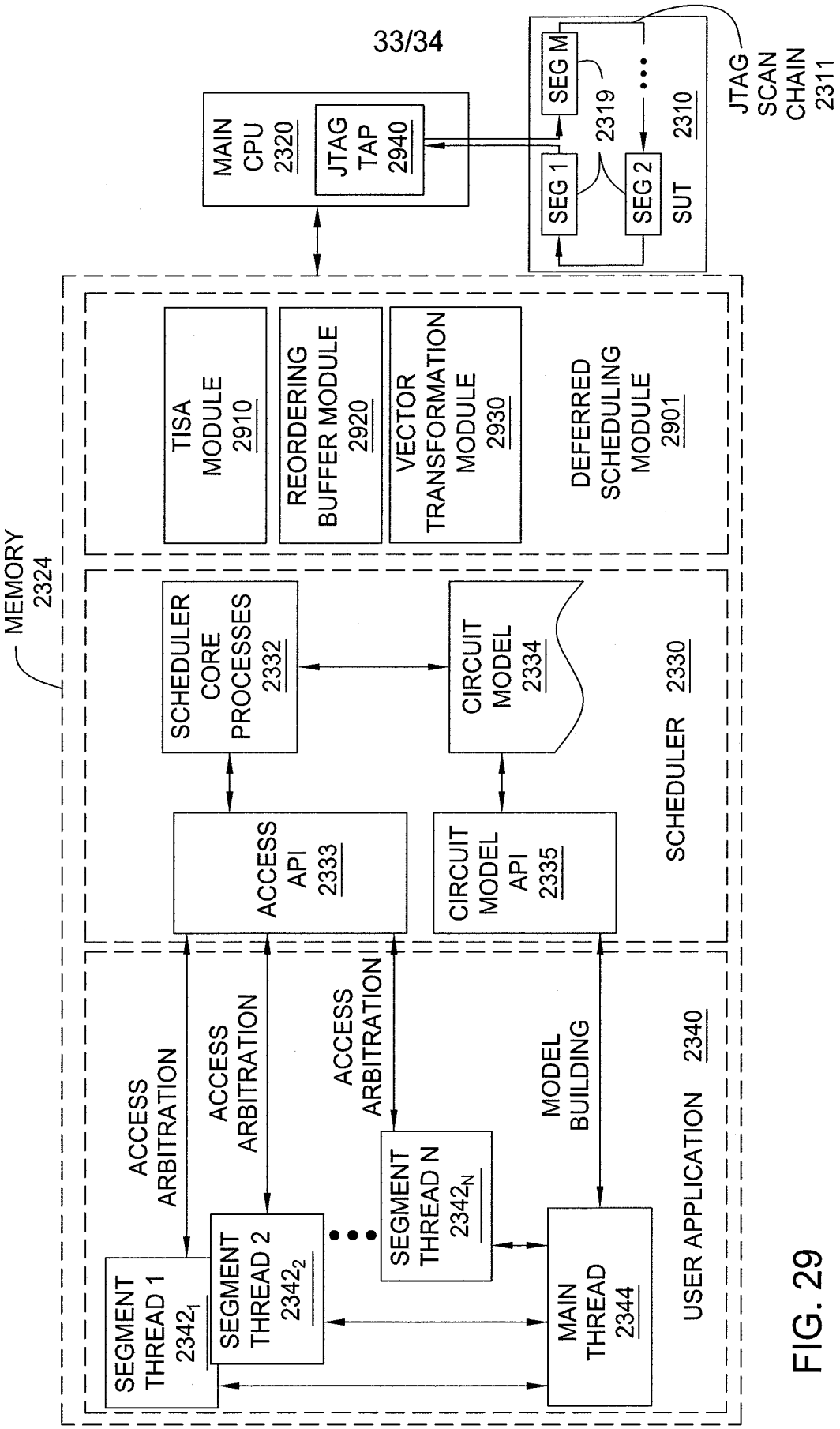
MEMORY 2324
MAIN CPU 2320
JTAG TAP 2940
SEG M
SEG 1
SEG 2
2319
SUT 2310
JTAG SCAN CHAIN 2311
TISA MODULE 2910
REORDERING BUFFER MODULE 2920
VECTOR TRANSFORMATION MODULE 2930
DEFERRED SCHEDULING MODULE 2901
SCHEDULER CORE PROCESSES 2332
CIRCUIT MODEL 2334
ACCESS API 2333
CIRCUIT MODEL API 2335
SCHEDULER 2330
ACCESS ARBITRATION
ACCESS ARBITRATION
ACCESS ARBITRATION
MODEL BUILDING
SEGMENT THREAD 1 2342$_1$
SEGMENT THREAD 2 2342$_2$
SEGMENT THREAD N 2342$_N$
MAIN THREAD 2344
USER APPLICATION 2340

FIG. 29

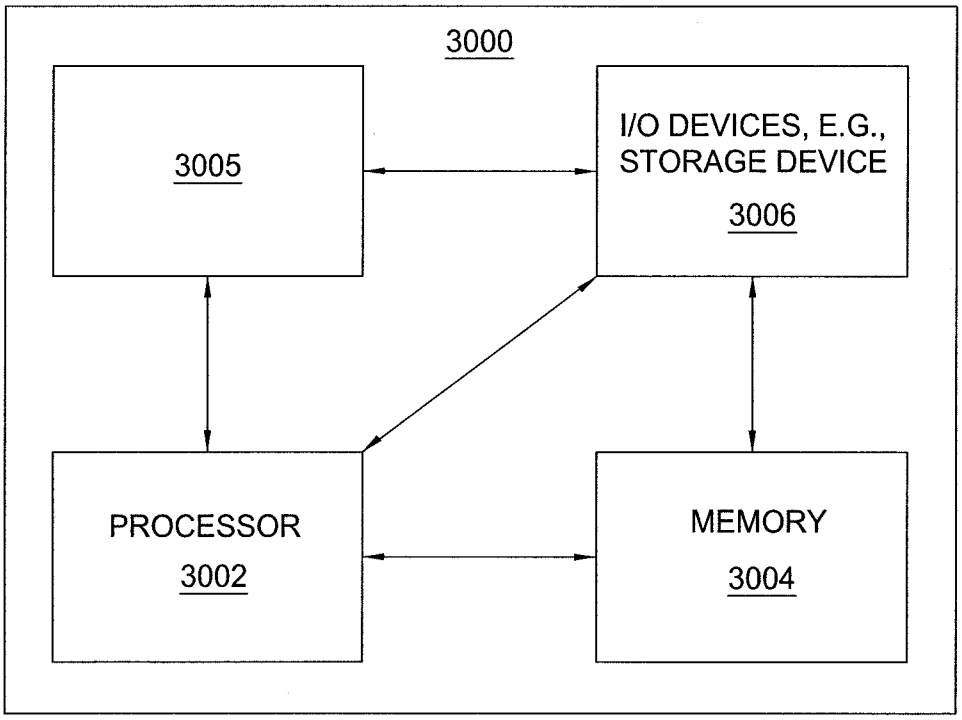
3000
3005
I/O DEVICES, E.G.,
STORAGE DEVICE
3006
PROCESSOR
3002
MEMORY
3004

FIG. 30

INTERNATIONAL SEARCH REPORT

International application No
PCT/US2012/061824

A. CLASSIFICATION OF SUBJECT MATTER
INV. G01R31/3183 G01R31/3185
ADD.

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
G01R

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

EPO-Internal , WPI Data, INSPEC

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	US 2010/229036 AI (GOYAL SURESH [US] ET AL) 9 September 2010 (2010-09-09) paragraph [0089] - paragraph [0238]	1-10
Y	KUCUK G ET AL: "Di stributed reorder buffer schemes for low power" , PROCEEDINGS 2003 I EEE INTERNATIONAL CONFERENCE ON COMPUTER DESIGN : VLSI I N C0MUTERS AND PROCESSORS. ICCD 2003. SAN JOSE, CA, OCT. 13 - 15, 2003 ; [INTERNATIONAL CONFERENCE ON COMPUTER DESIGN] , LOS ALAMIT0S, CA, I EEE C0MP. S0C, US, 13 October 2003 (2003-10-13) , pages 364-370, XP010664113 , D0I : 10. 1109/ICCD. 2003. 1240920 ISBN : 978-0-7695-2025-4 the whole document	1-10

☐ Further documents are listed in the continuation of Box C.

☒ See patent family annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier application or patent but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) orwhich is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
12 Apri l 2013	19/04/2013

Name and mailing address of the ISA/	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Fax: (+31-70) 340-3016	Meggyesi , Zol tan

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No
PCT/US2012/061824

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2010229036 Al	09-09-2010	CN 102341718 A	01-02-2012
		CN 102341719 A	01-02-2012
		CN 102439470 A	02-05-2012
		EP 2404182 Al	11-01-2012
		EP 2404183 Al	11-01-2012
		EP 2404184 Al	11-01-2012
		JP 2012519853 A	30-08-2012
		JP 2012519854 A	30-08-2012
		JP 2012519912 A	30-08-2012
		KR 20110122165 A	09-11-2011
		KR 20110124274 A	16-11-2011
		KR 20120023601 A	13-03-2012
		US 2010229036 Al	09-09-2010
		US 2010229042 Al	09-09-2010
		US 2010229058 Al	09-09-2010
		WO 2010101984 Al	10-09-2010
		WO 2010101995 Al	10-09-2010
		WO 2010102019 Al	10-09-2010