US 20160070662A1

(19) **United States**

(12) **Patent Application Publication**

Ly et al.

(10) **Pub. No.: US 2016/0070662 A1**

(43) **Pub. Date: Mar. 10, 2016**

(54) **REORDERING A SEQUENCE OF MEMORY ACCESSES TO IMPROVE PIPELINED PERFORMANCE**

(71) Applicant: **NATIONAL INSTRUMENTS CORPORATION**, Austin, TX (US)

(72) Inventors: **Tai A. Ly**, Austin, TX (US); **Swapnil D. Mhaske**, Highland Park, NJ (US); **Hojin Kee**, Austin, TX (US); **Adam T. Arnesen**, Pflugerville, TX (US); **David C. Uliana**, Austin, TX (US); **Newton G. Petersen**, Emporia, KS (US)

(21) Appl. No.: **14/523,232**

(22) Filed: **Oct. 24, 2014**

Related U.S. Application Data

(60) Provisional application No. 62/045,832, filed on Sep. 4, 2014.

Publication Classification

(51) **Int. Cl.**

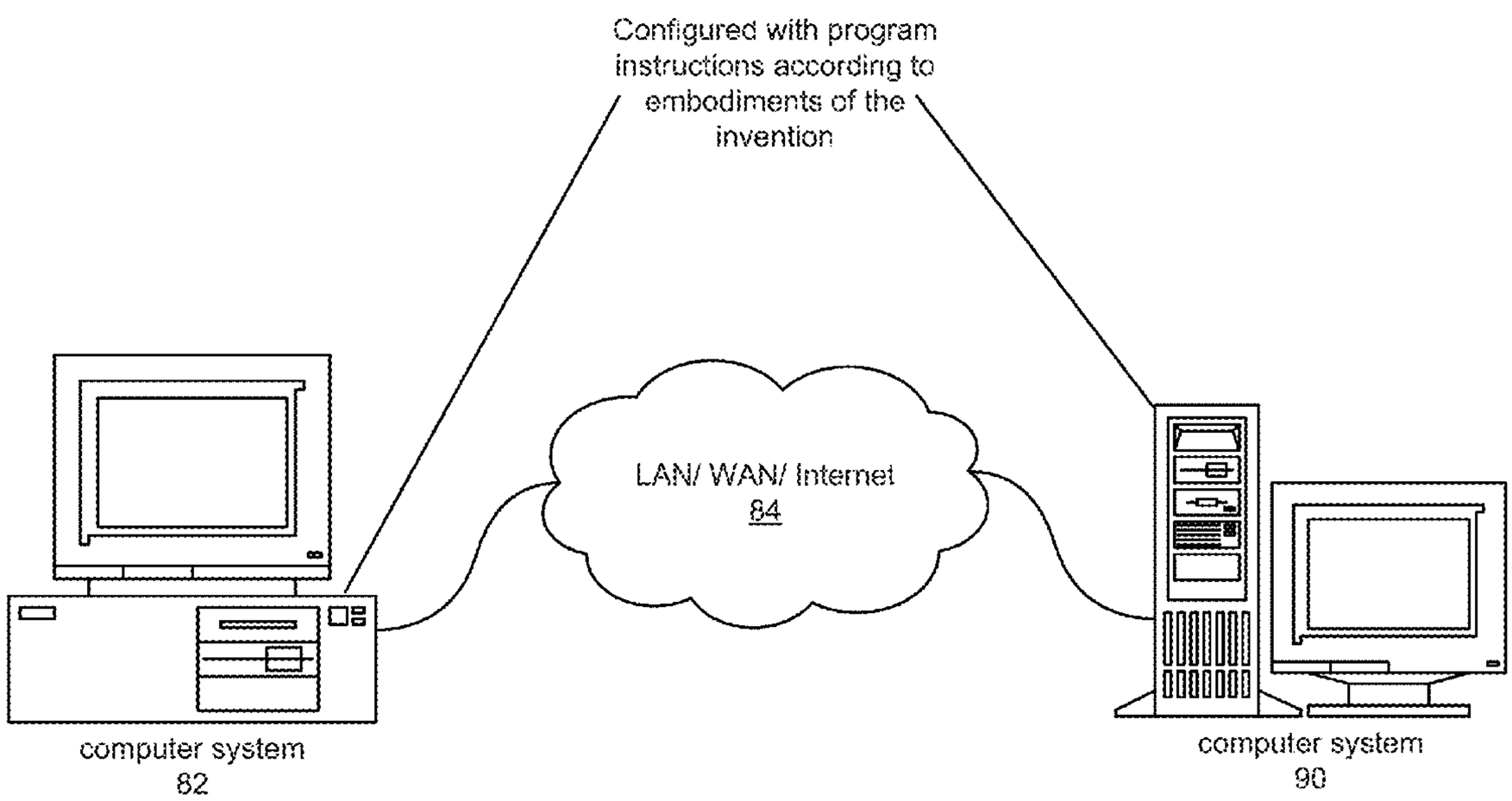
G06F 13/16	(2006.01)
G11C 7/10	(2006.01)

(52) **U.S. Cl.**

CPC ***G06F 13/1626*** (2013.01); ***G11C 7/1039*** (2013.01); ***G06F 13/1615*** (2013.01)

(57) **ABSTRACT**

Techniques are disclosed relating to reordering sequences of memory accesses. In one embodiment, a method includes storing a specified sequence of memory accesses that corresponds to a function to be performed. In this embodiment, the specified sequence of memory accesses has first memory access constraints. In this embodiment, the method further includes reordering the specified sequence of memory accesses to create a reordered sequence of memory accesses that has second, different memory access constraints. In this embodiment, the reordered sequence of memory accesses is usable to access a memory to perform the function. In some embodiments, performance estimates are determined for a plurality of reordered sequences of memory accesses, and one of the reordered sequences is selected based on the performance estimates. In some embodiments, the reordered sequence is used to compile a program usable to perform the function.



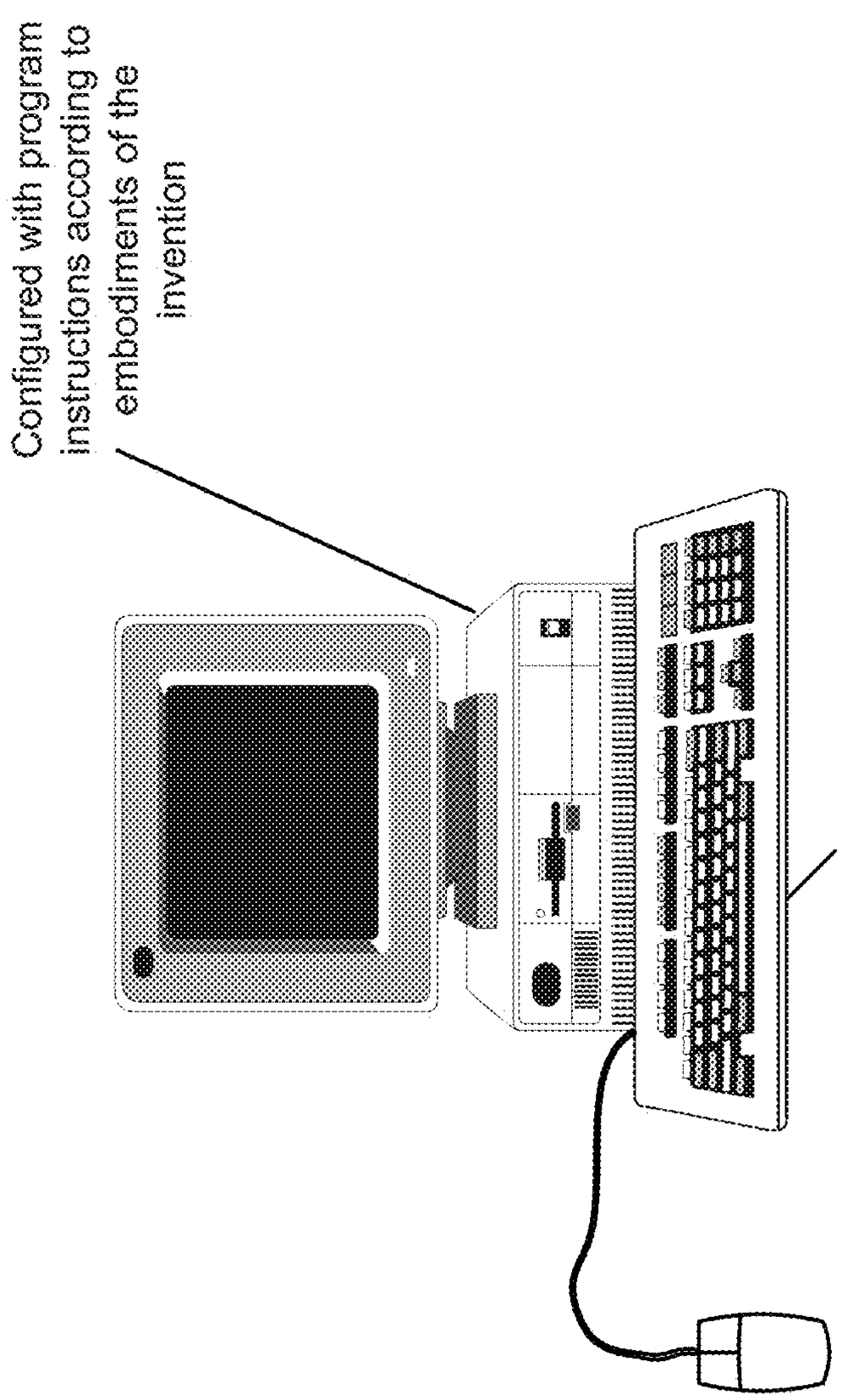

Configured with program instructions according to embodiments of the invention
Computer System 82

Figure 1A

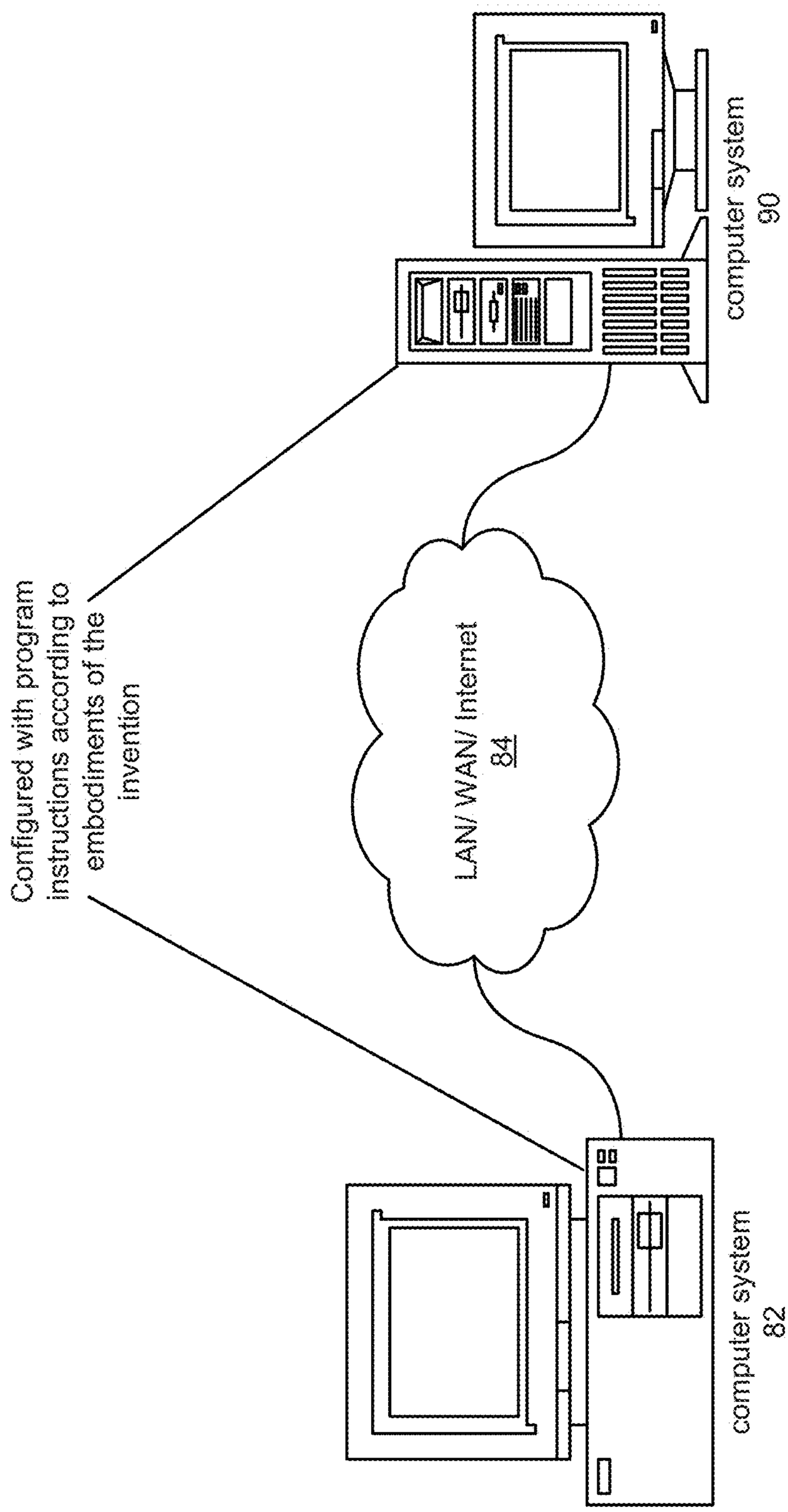

computer system
90
Configured with program instructions according to embodiments of the invention
LAN/ WAN/ Internet
84
computer system
82

Figure 1B

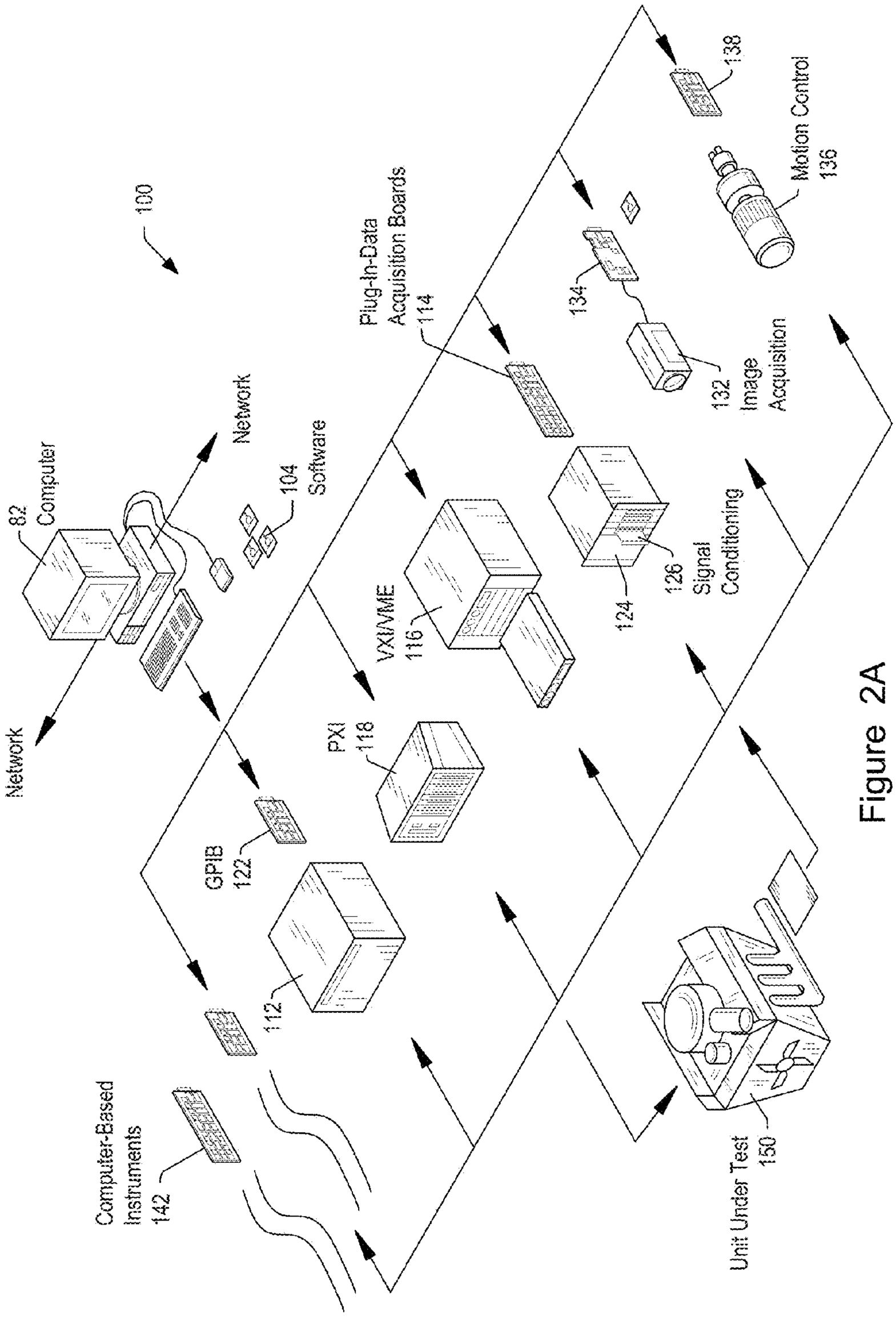

100
82
Computer
Network
Network
104
Software
Plug-In-Data
Acquisition Boards
114
138
Motion Control
136
134
132
Image
Acquisition
VXI/VME
116
124
126
Signal
Conditioning
PXI
118
GPIB
122
112
Computer-Based
Instruments
142
Unit Under Test
150

Figure 2A

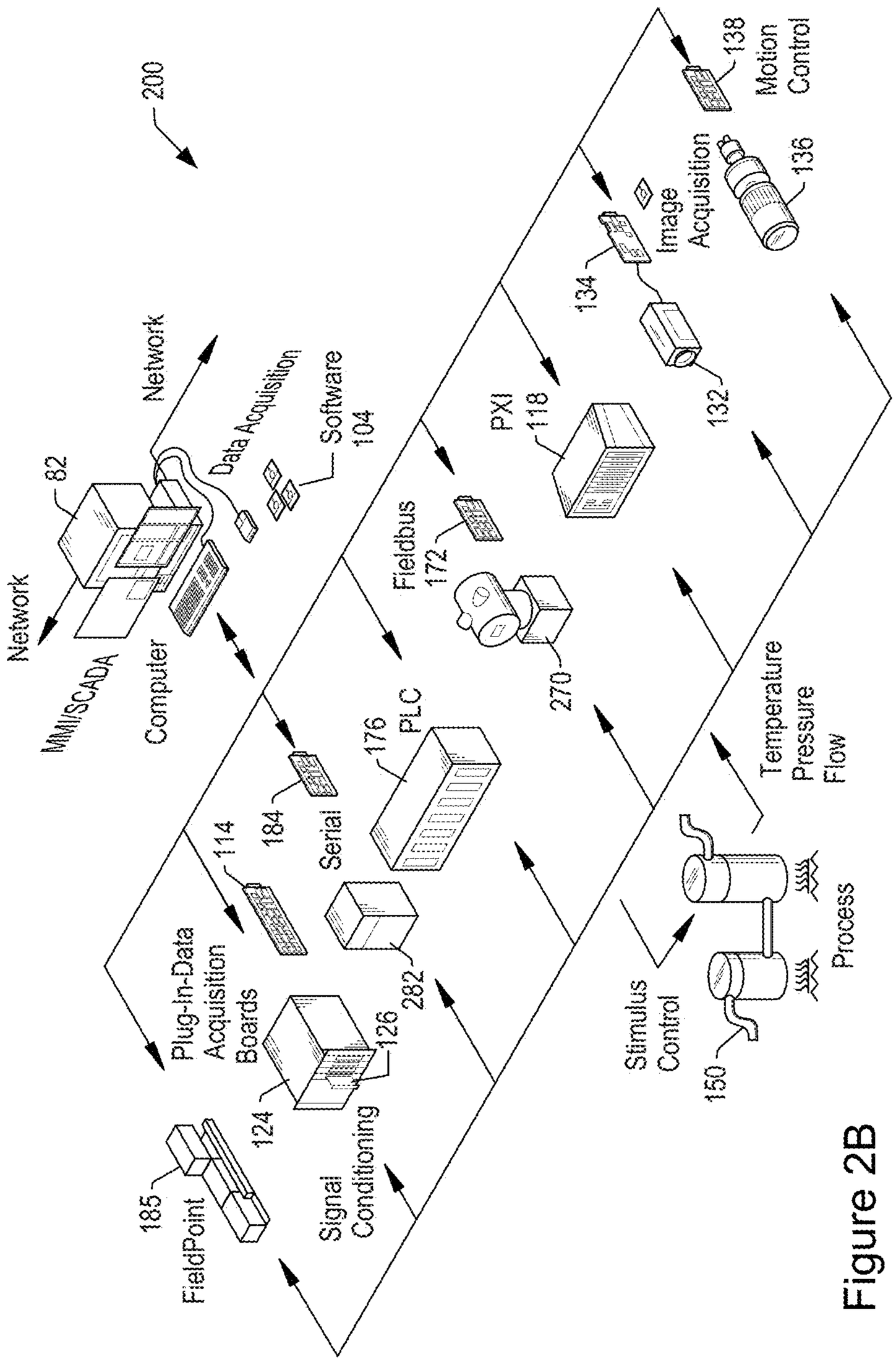

200
138
Motion Control
Image Acquisition
136
134
132
Network
82
MMI/SCADA
Computer
Data Acquisition Software
104
PXI
118
Fieldbus
172
270
176
PLC
184
Serial
114
Plug-In-Data Acquisition Boards
282
126
124
Signal Conditioning
185
FieldPoint
Temperature
Pressure
Flow
Process
Stimulus Control
150

Figure 2B

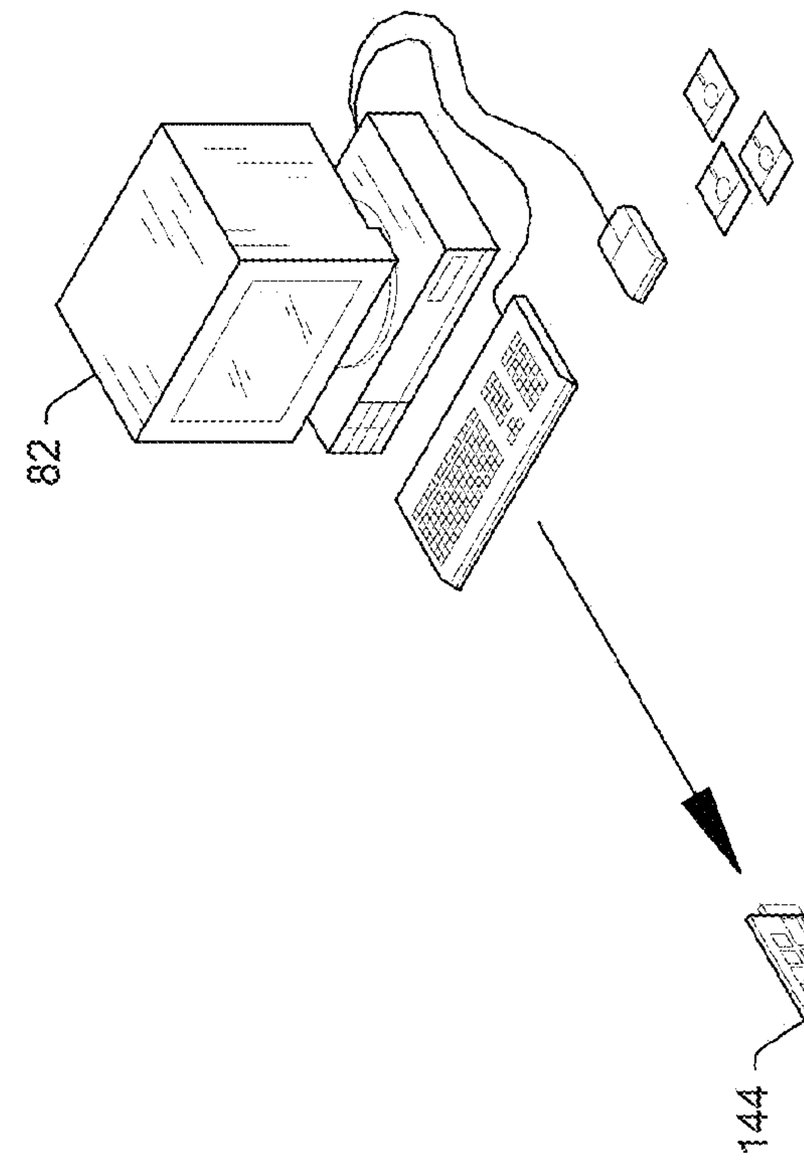


Figure 3B

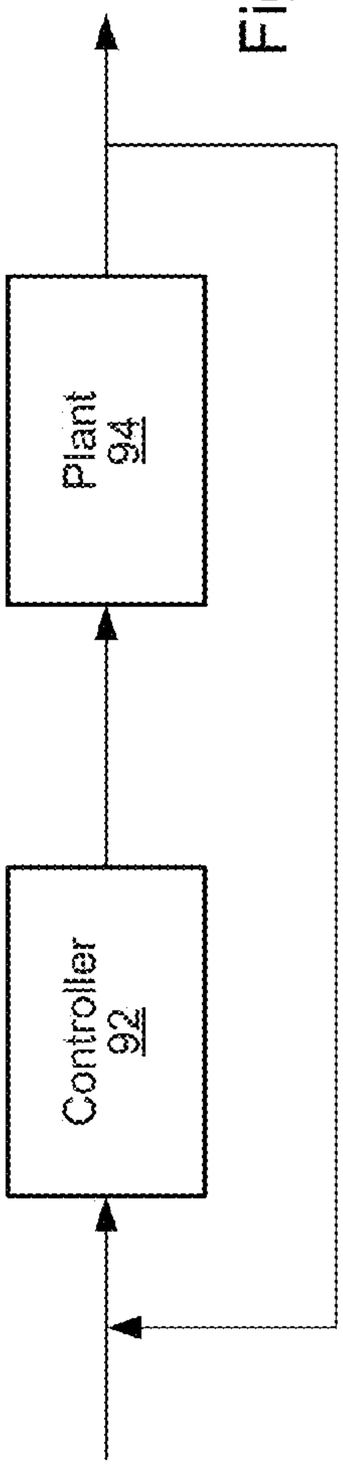


Figure 3A

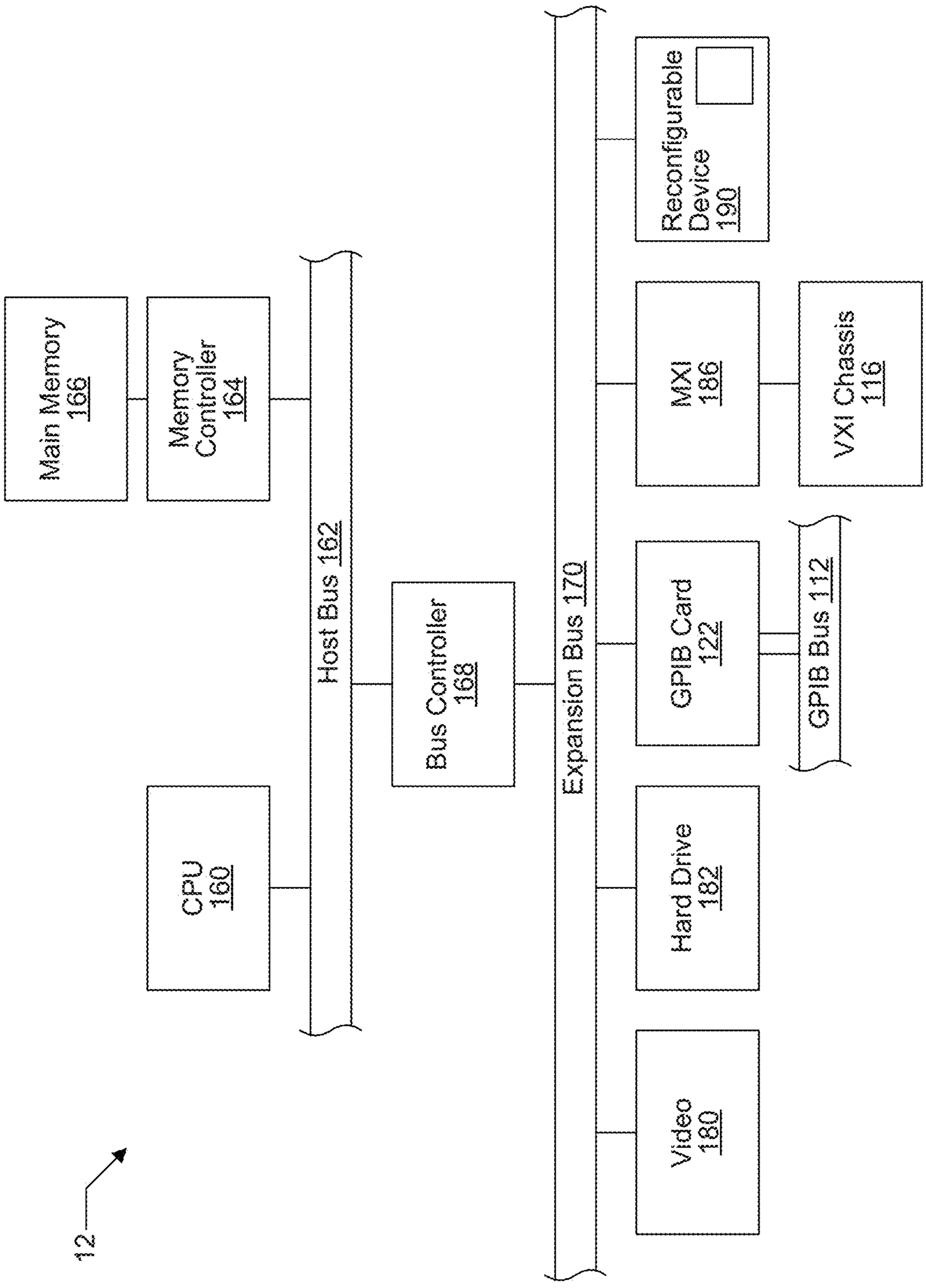

Reconfigurable Device 190
Main Memory 166
Memory Controller 164
Host Bus 162
Bus Controller 168
Expansion Bus 170
MXI 186
VXI Chassis 116
GPIB Card 122
GPIB Bus 112
CPU 160
Hard Drive 182
Video 180
12

Figure 4

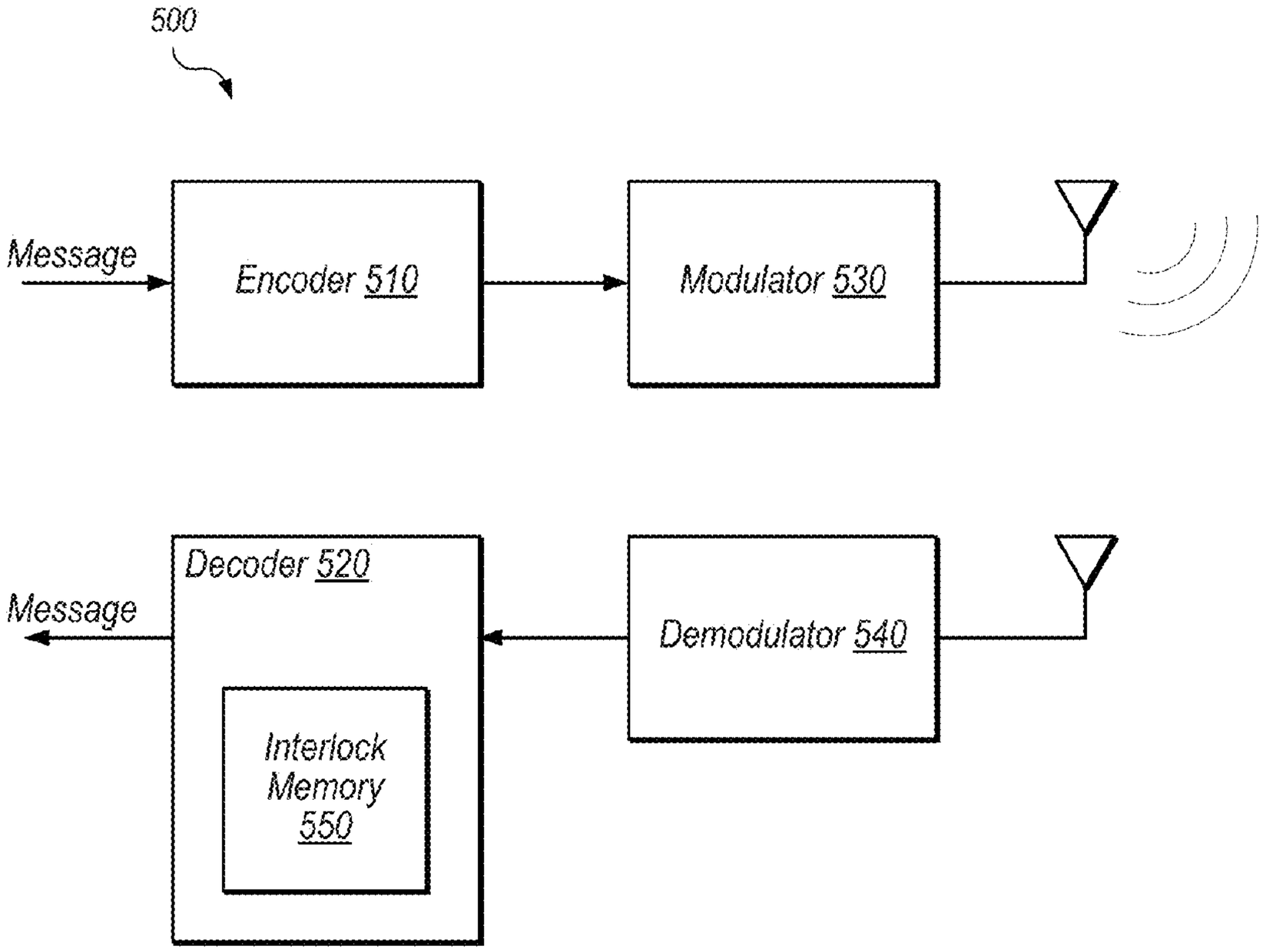

500
Message
Encoder 510
Modulator 530
Decoder 520
Message
Interlock Memory 550
Demodulator 540

Figure 5

H Matrix 620

Layer ↓ Block →

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
L1	57				50		11		50		79		1	0										
L2	3		28		0				55	7				0	0									
L3	30				24	37			56	14					0	0								
L4	62	53			53			3	35							0	0							
L5	40			20	66			22	28								0	0						
L6	0				8		42		50			8						0	0					
L7	69	79	79				56		52				0						0	0				
L8	65				38	57			72		27									0	0			
L9	64				14	52			30			32									0	0		
L10		45		70	0				77	9												0	0	
L11	2	56		57	35						12												0	0
L12	24		61		60			27	51			16	1											0

Figure 6A

Beta Matrix 640

Layer ↓ Block Index

L1	0	4	6	8	10	12	13	
L2	0	2	4	8	9	13	14	
L3	0	4	5	8	9	14	15	
L4	0	1	4	7	8	15	16	
L5	0	3	4	7	8	16	17	
L6	0	4	6	8	11	17	18	
L7	0	1	2	6	8	12	18	19
L8	0	4	5	8	10	19	20	
L9	0	4	5	8	11	20	21	
L10	1	3	4	8	9	21	22	
L11	0	1	3	4	10	22	23	
L12	0	2	4	7	8	11	12	23

Figure 6B

Rearranged Beta Matrix 660

Layer ↓ Block Index

L1	**0**	4	6	13	6	10	12	
L2	9	**0**	4	8	13	14	2	
L3	15	9	**0**	4	8	5	14	
L4	7	15	16	**0**	4	8	1	
L5	17	7	3	16	**0**	4	8	
L6	6	17	18	11		**0**	4	8
L7	19	6	0	8	1	2	16	12
L8	4	19	5	0	8	20	10	
L9	21	4	11	5	0	8	20	
L10	1	21	4	3	22	9	8	
L11	0	1	23	4	3	22	10	
L12	8	0	2	23	4	12	7	11

Figure 6C

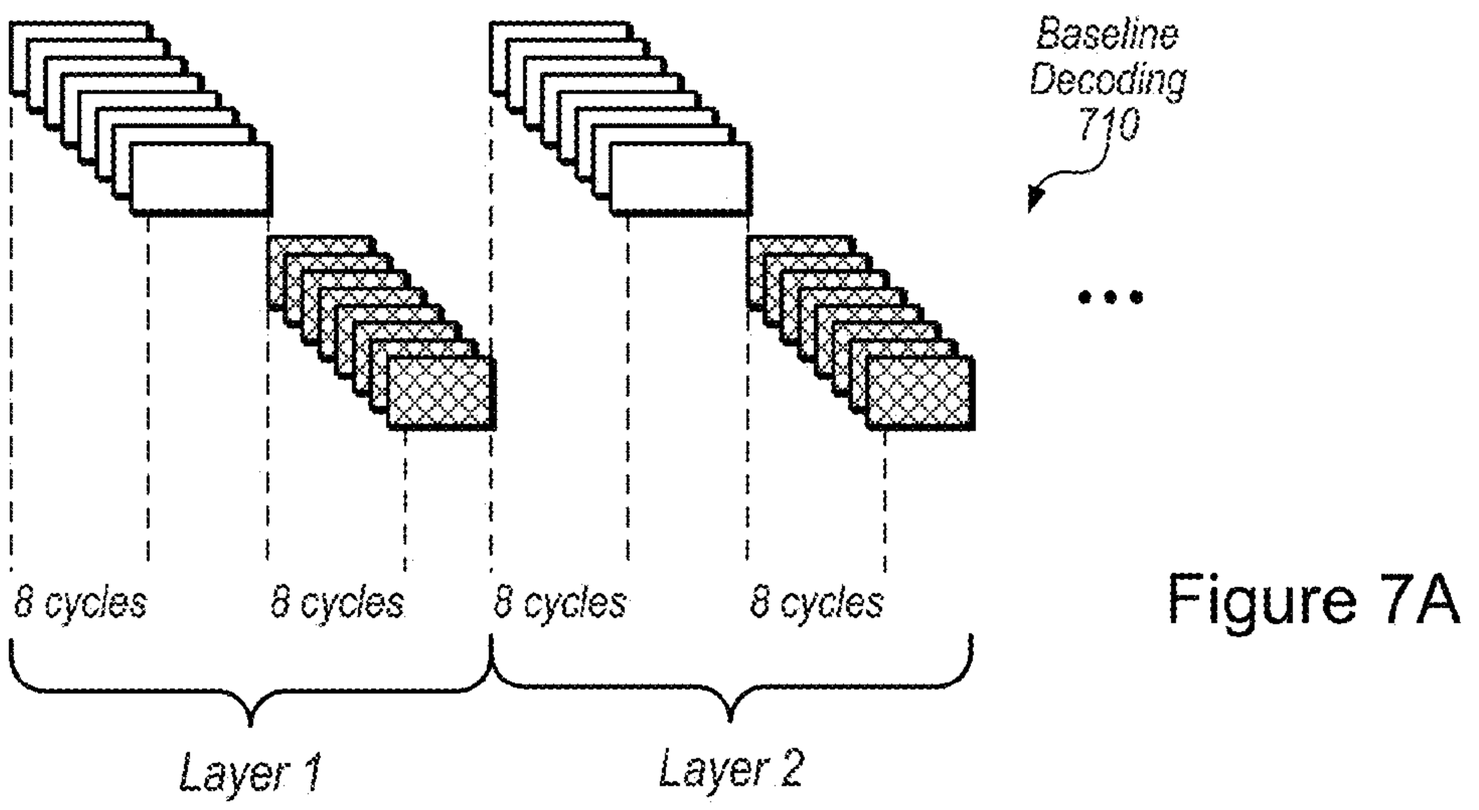


Figure 7A

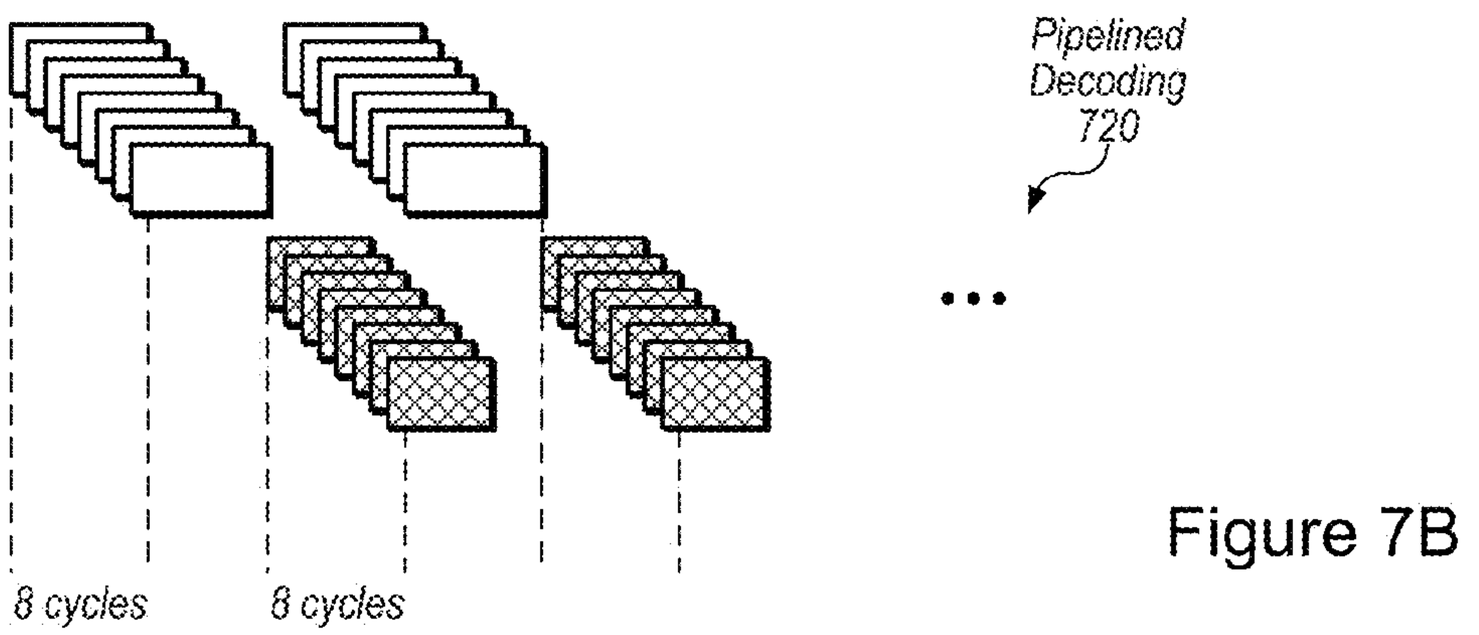


Figure 7B

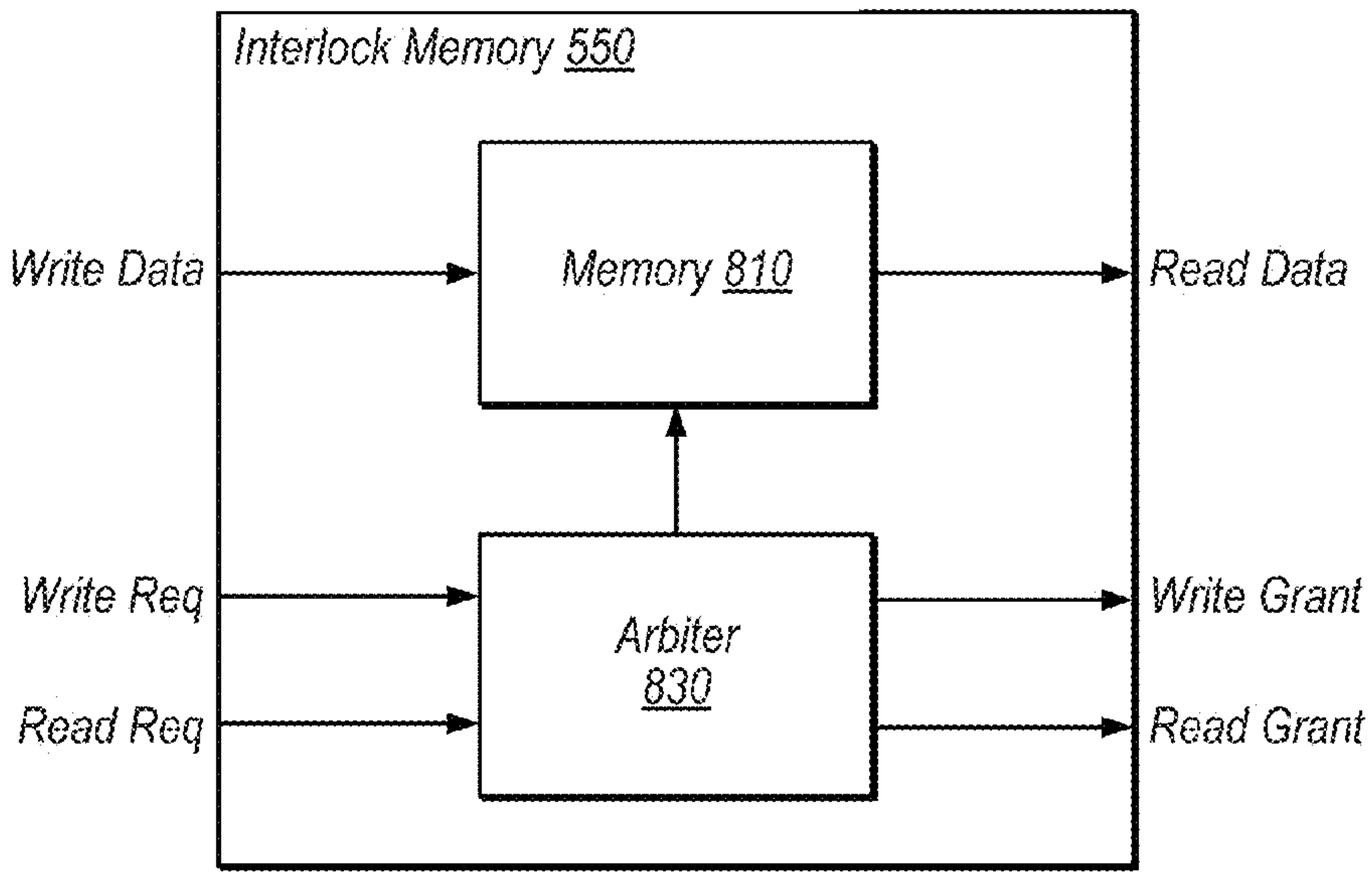


Figure 8A

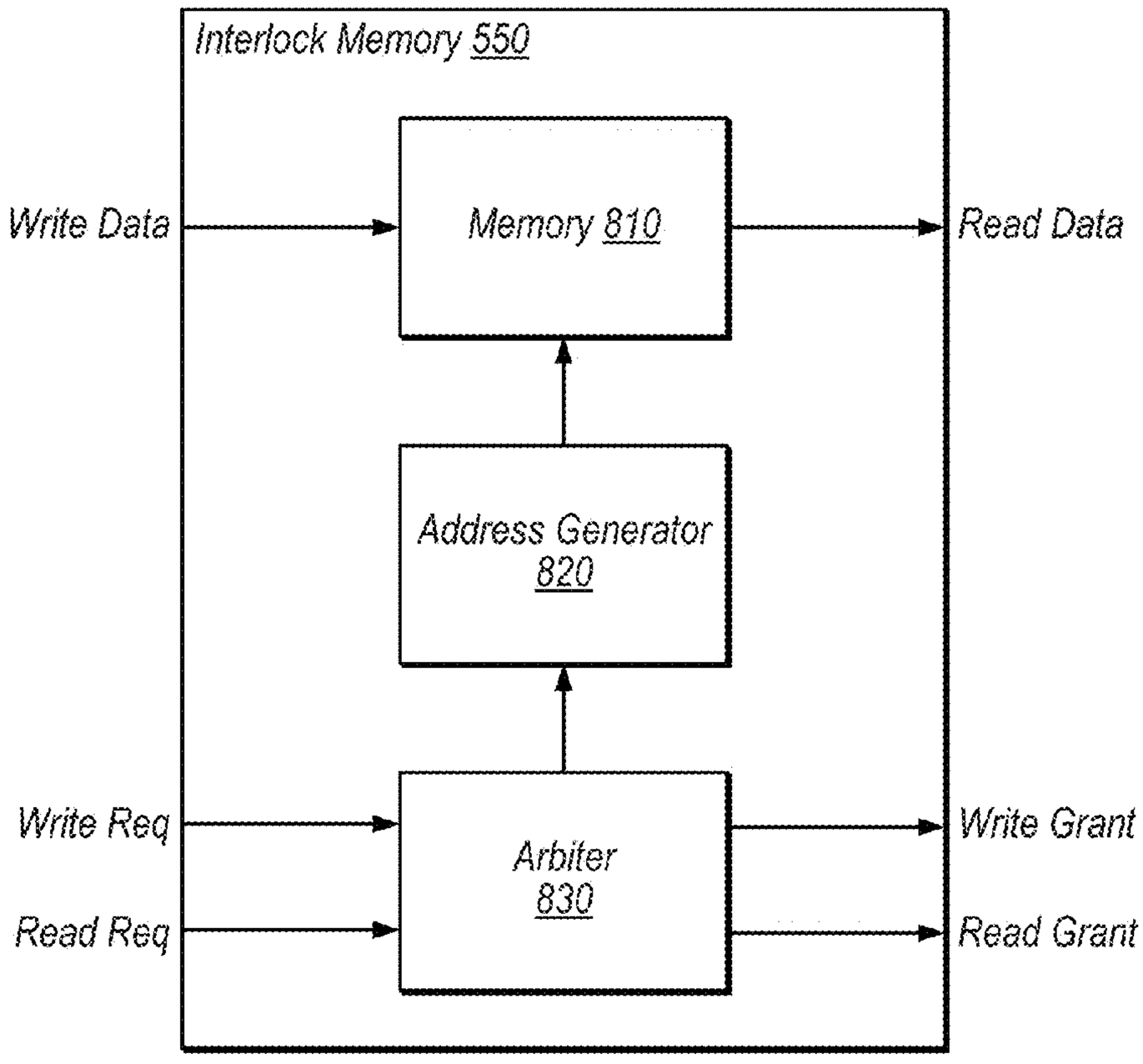


Figure 8B

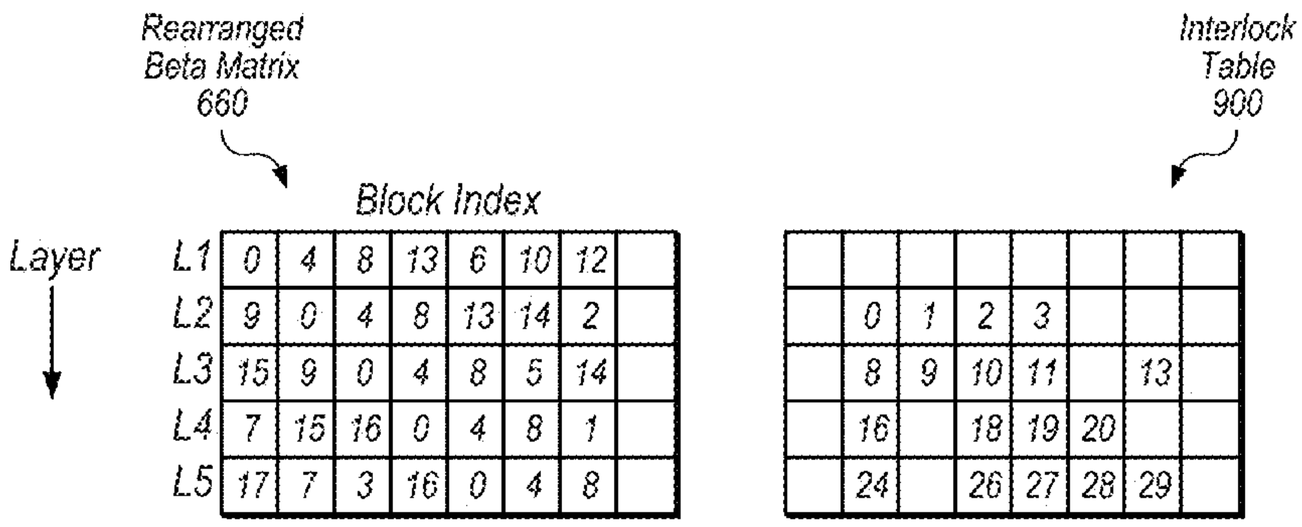


Figure 9

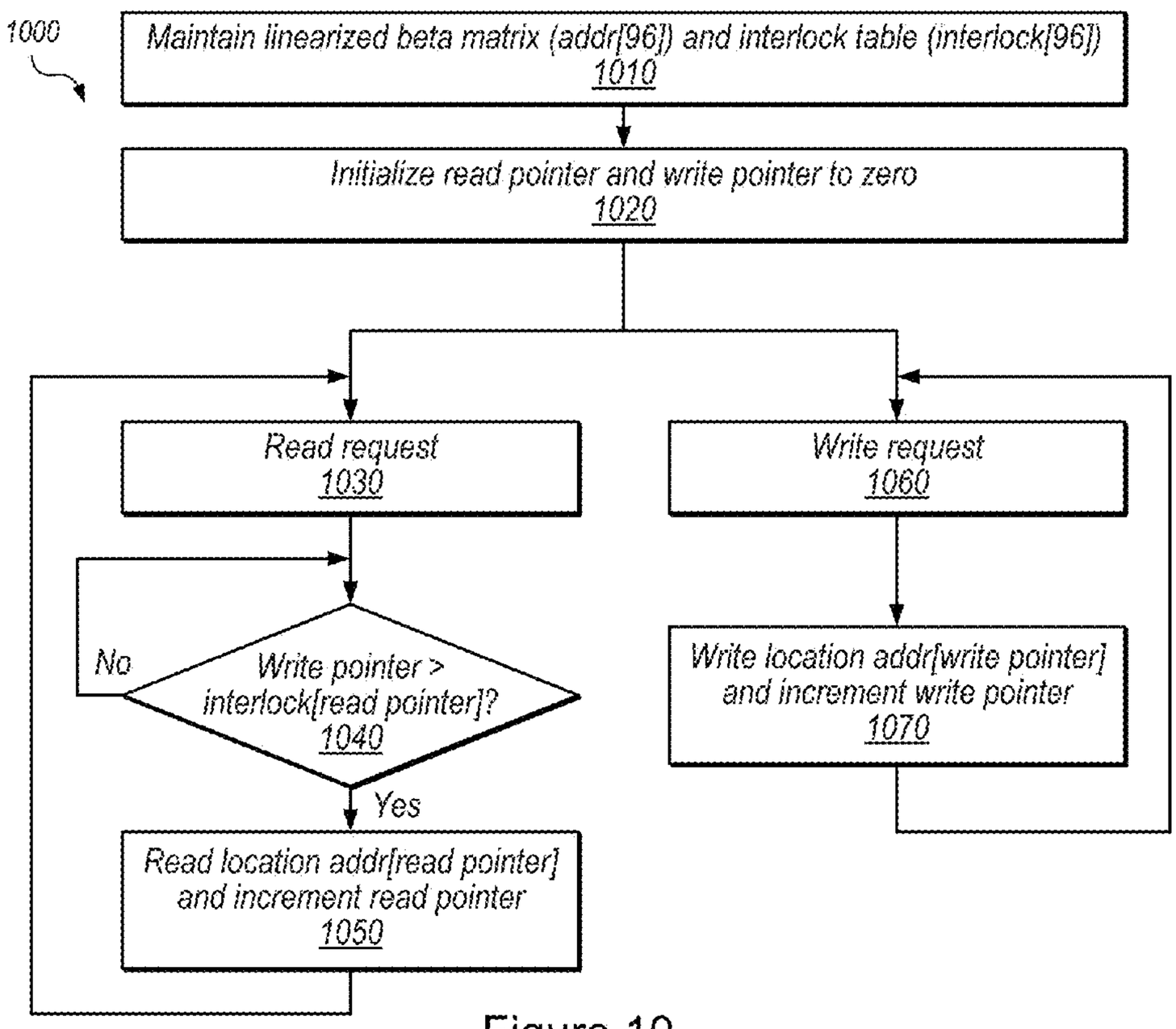


Figure 10

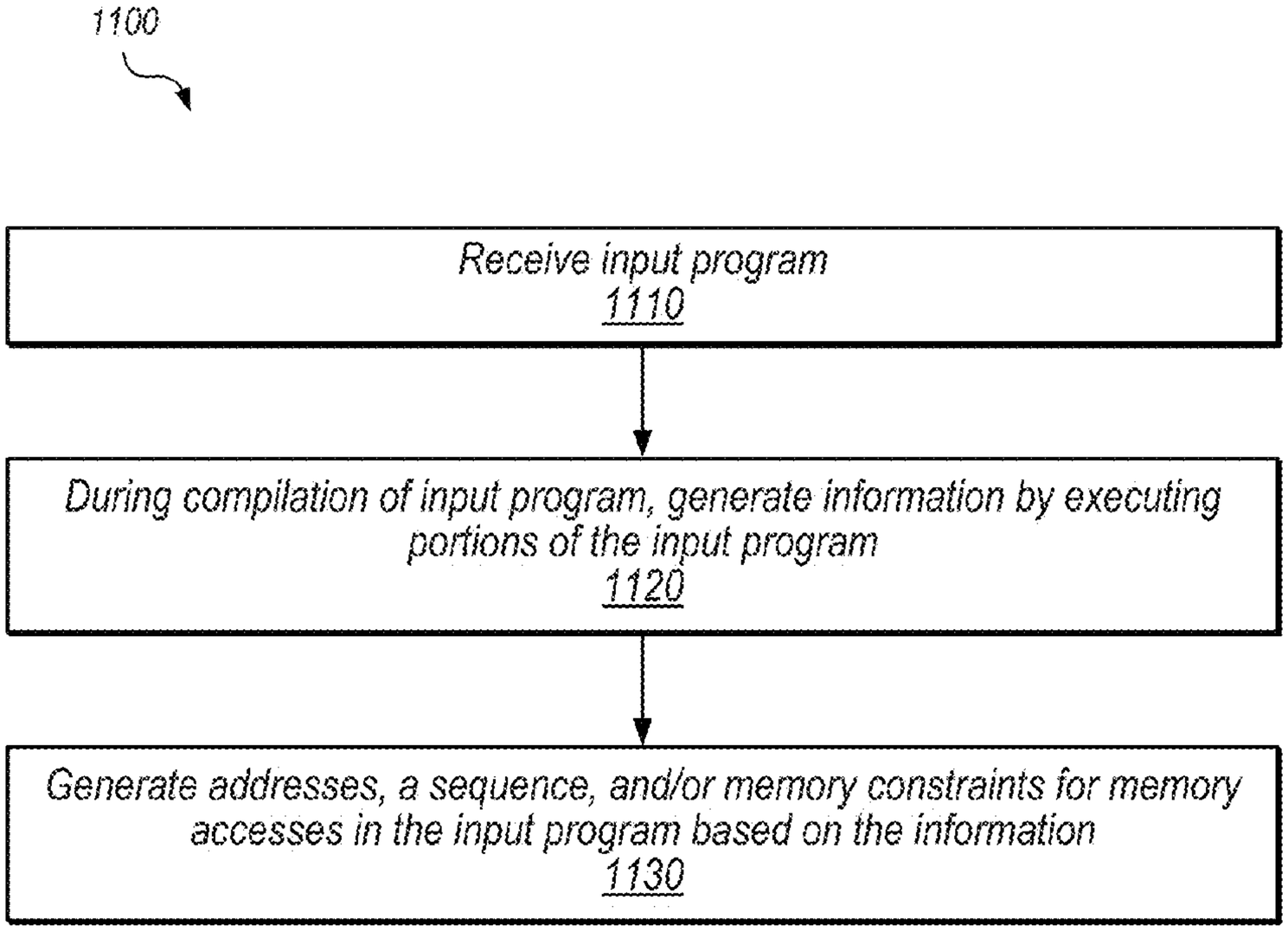

1100
Receive input program
1110
During compilation of input program, generate information by executing portions of the input program
1120
Generate addresses, a sequence, and/or memory constraints for memory accesses in the input program based on the information
1130

Figure 11

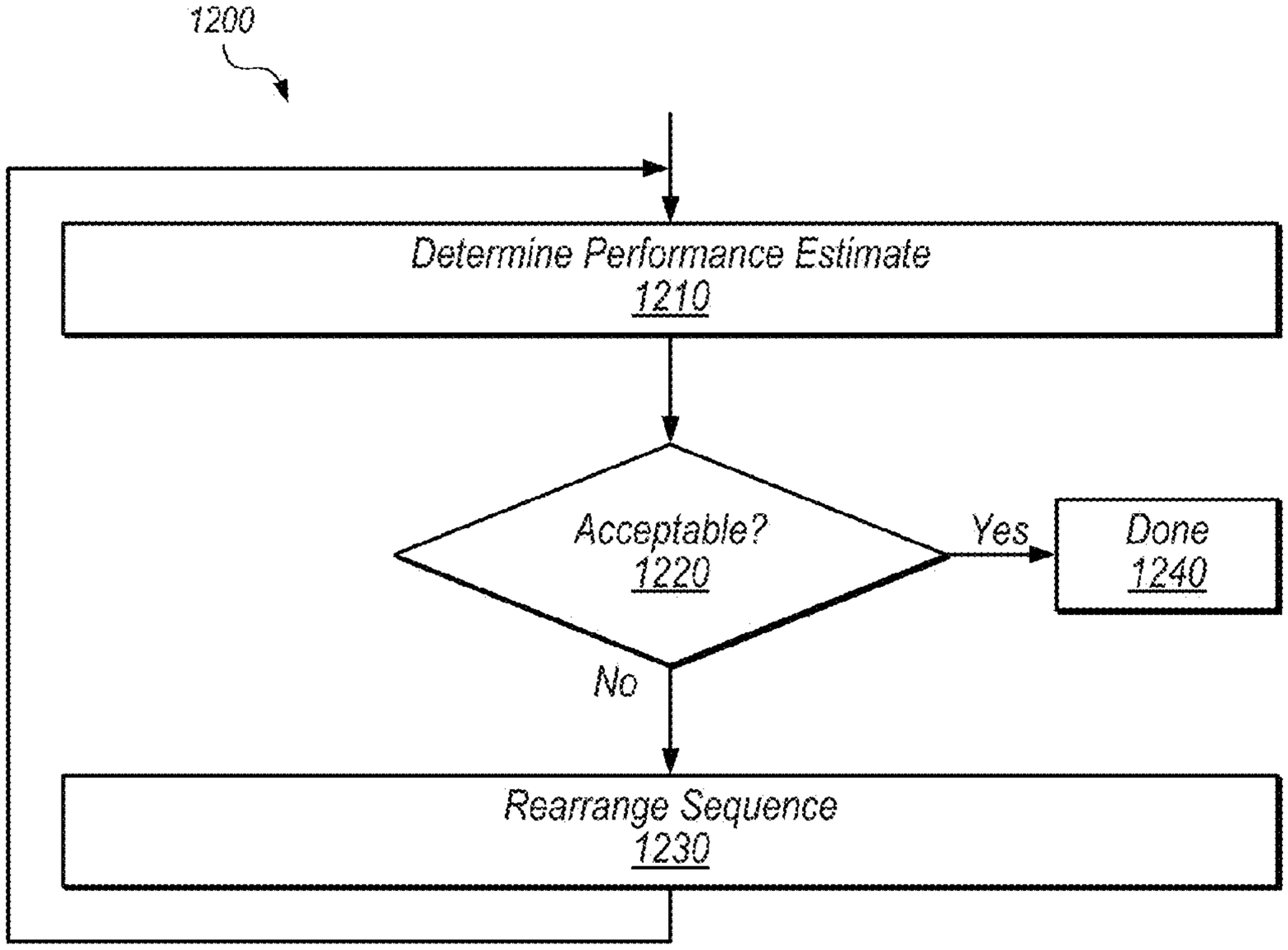

1200
Determine Performance Estimate
1210
Acceptable?
1220
Yes
Done
1240
No
Rearrange Sequence
1230

Figure 12

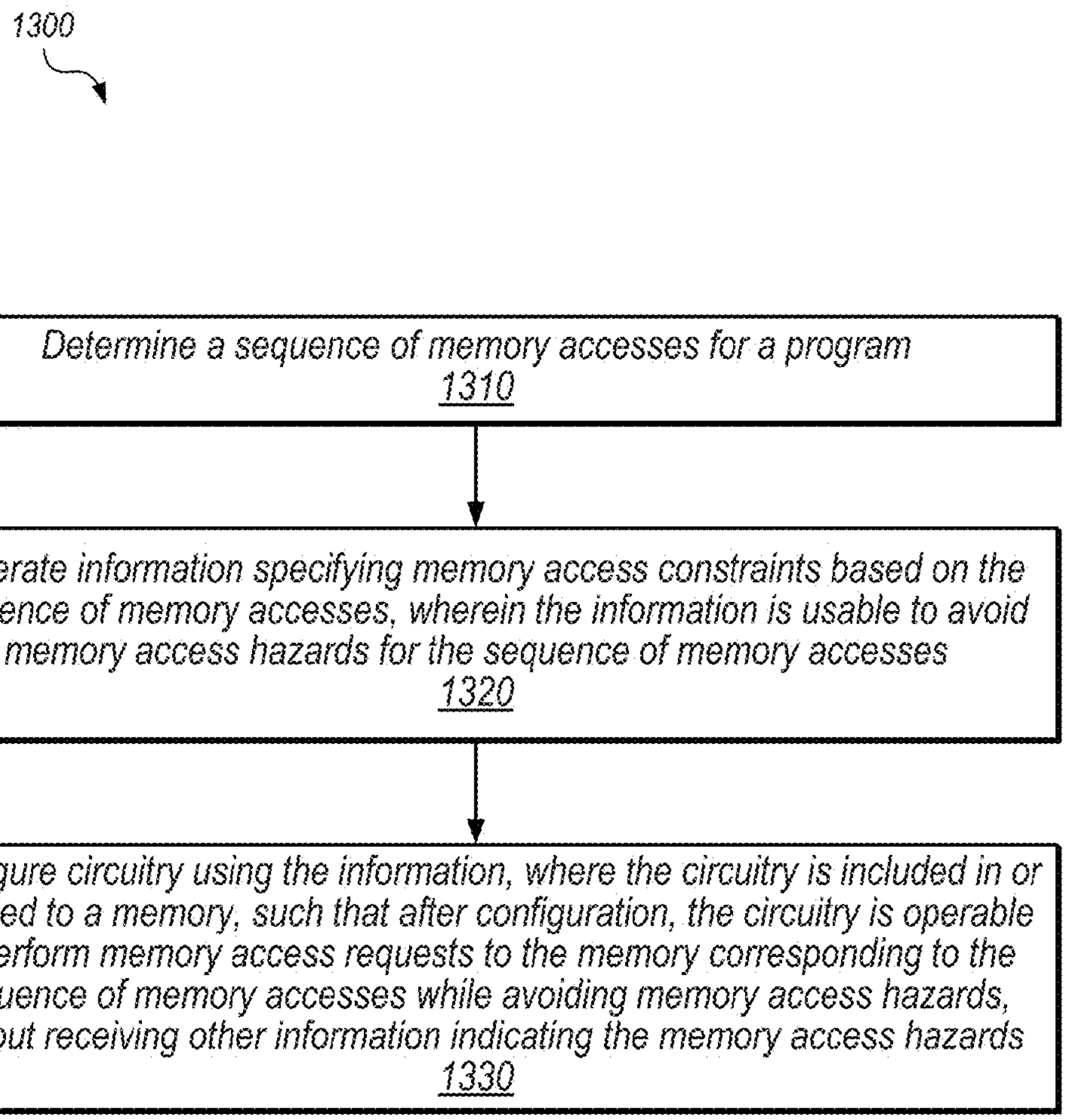

1300
Determine a sequence of memory accesses for a program
1310
Generate information specifying memory access constraints based on the sequence of memory accesses, wherein the information is usable to avoid memory access hazards for the sequence of memory accesses
1320
Configure circuitry using the information, where the circuitry is included in or coupled to a memory, such that after configuration, the circuitry is operable to perform memory access requests to the memory corresponding to the sequence of memory accesses while avoiding memory access hazards, without receiving other information indicating the memory access hazards
1330

Figure 13

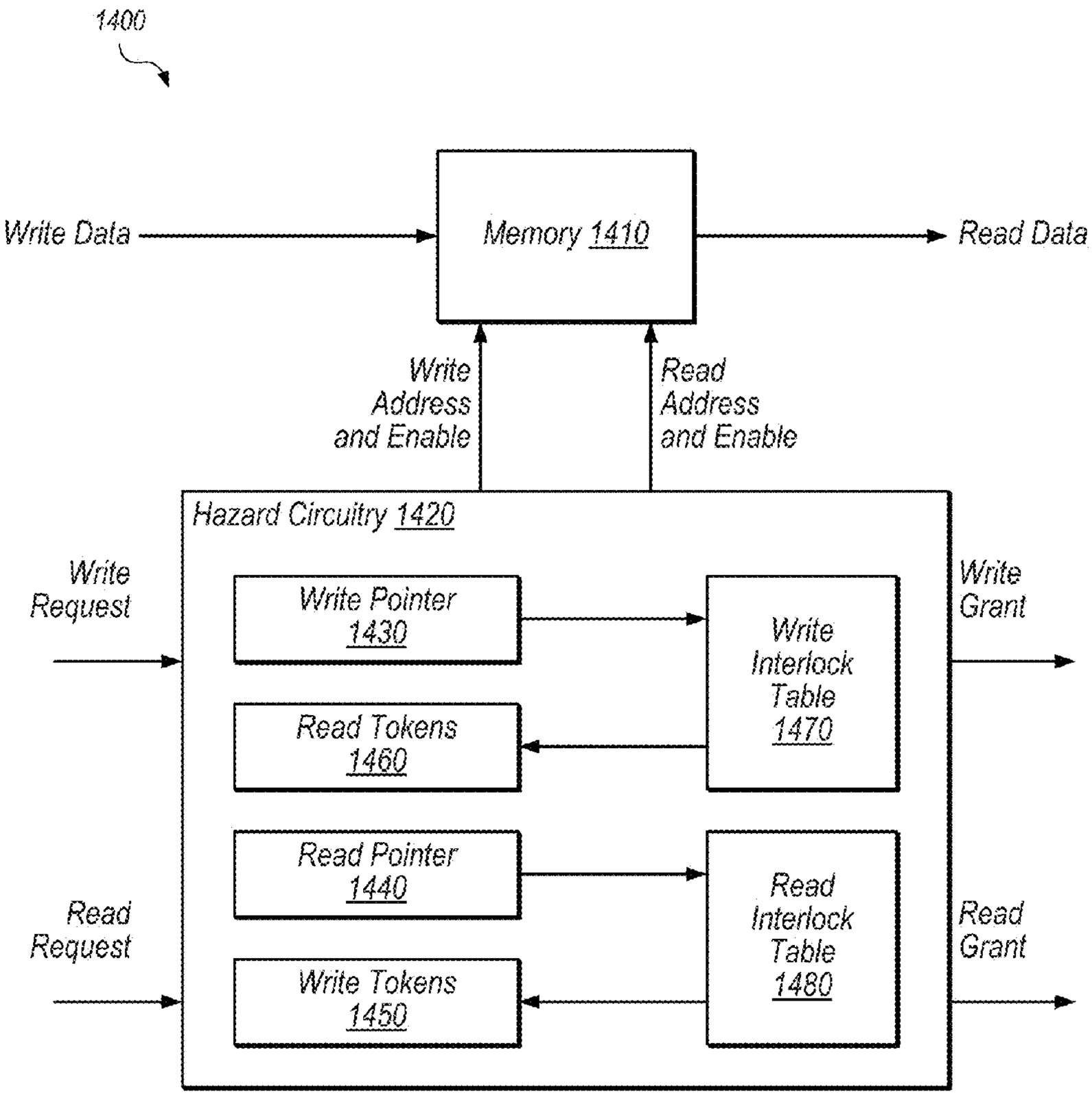

1400
Write Data
Memory 1410
Read Data
Write Address and Enable
Read Address and Enable
Hazard Circuitry 1420
Write Request
Write Pointer 1430
Write Interlock Table 1470
Write Grant
Read Tokens 1460
Read Pointer 1440
Read Request
Read Interlock Table 1480
Read Grant
Write Tokens 1450

Figure 14

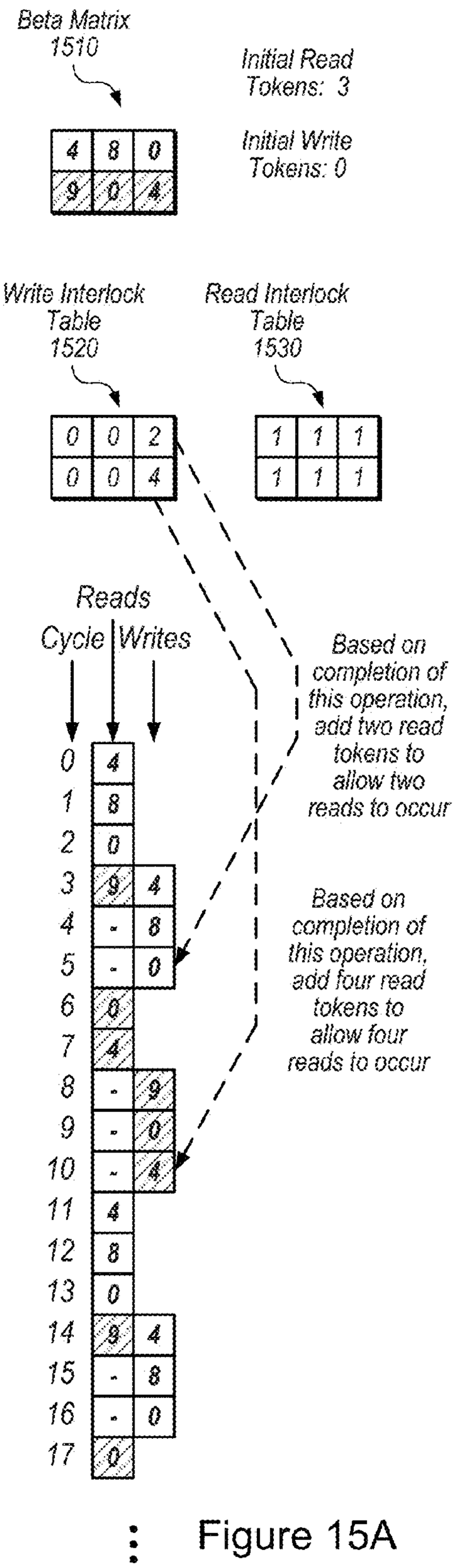


Figure 15A

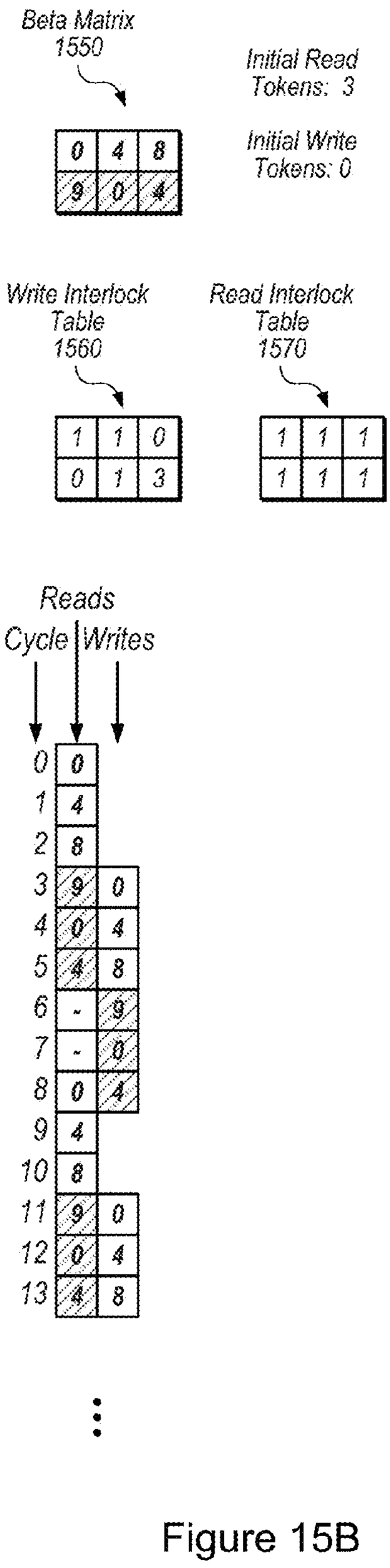


Figure 15B

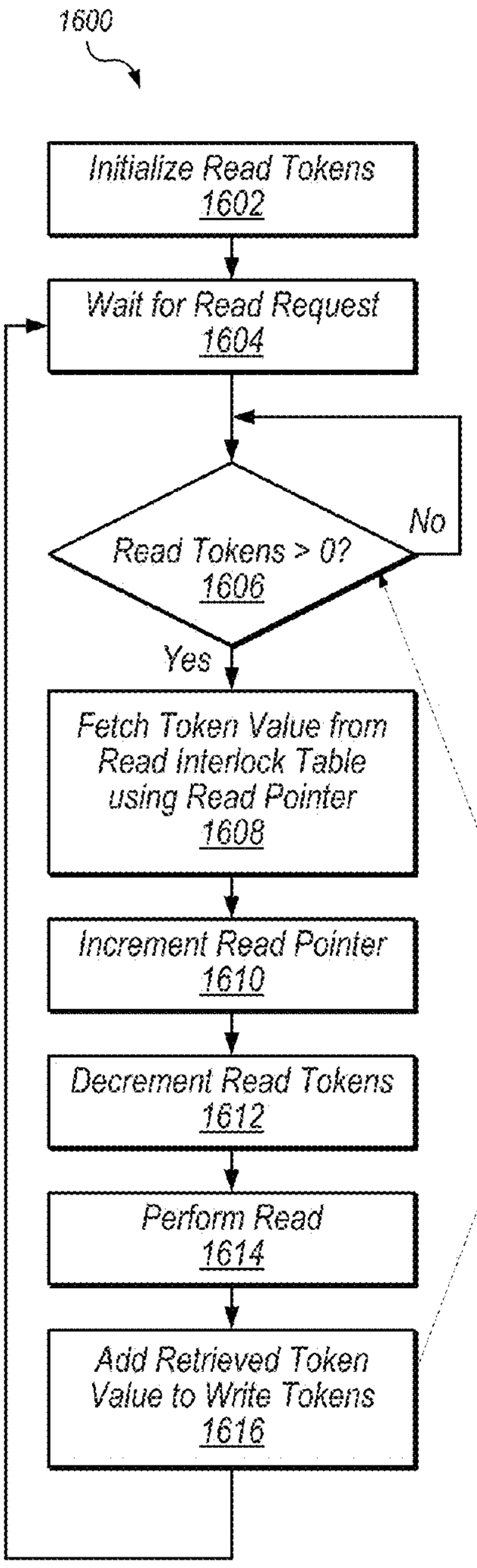


Figure 16A

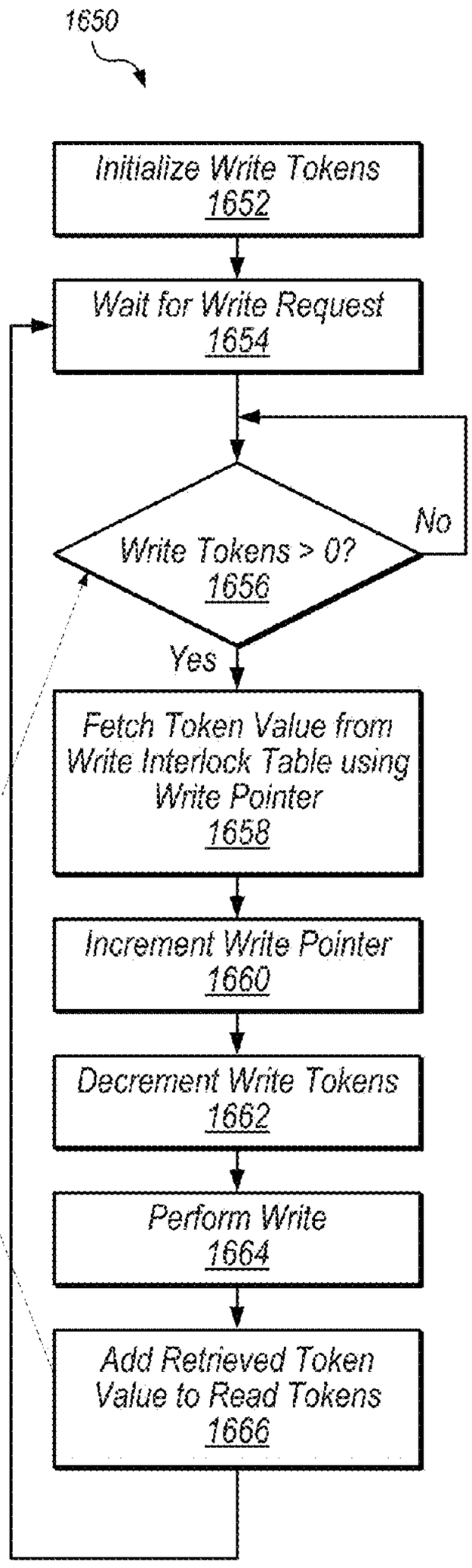


Figure 16B

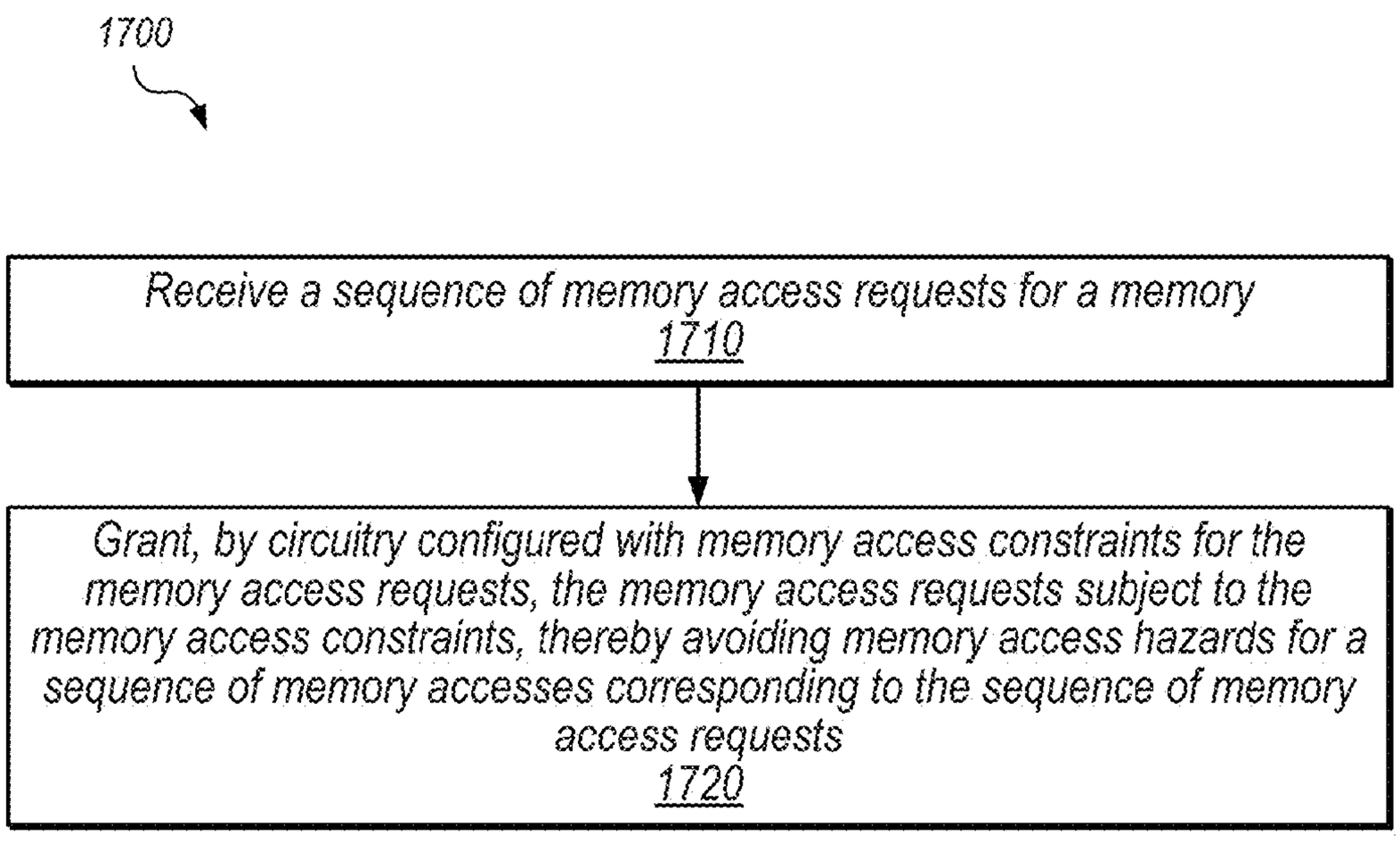

1700
Receive a sequence of memory access requests for a memory
1710
Grant, by circuitry configured with memory access constraints for the memory access requests, the memory access requests subject to the memory access constraints, thereby avoiding memory access hazards for a sequence of memory accesses corresponding to the sequence of memory access requests
1720

Figure 17

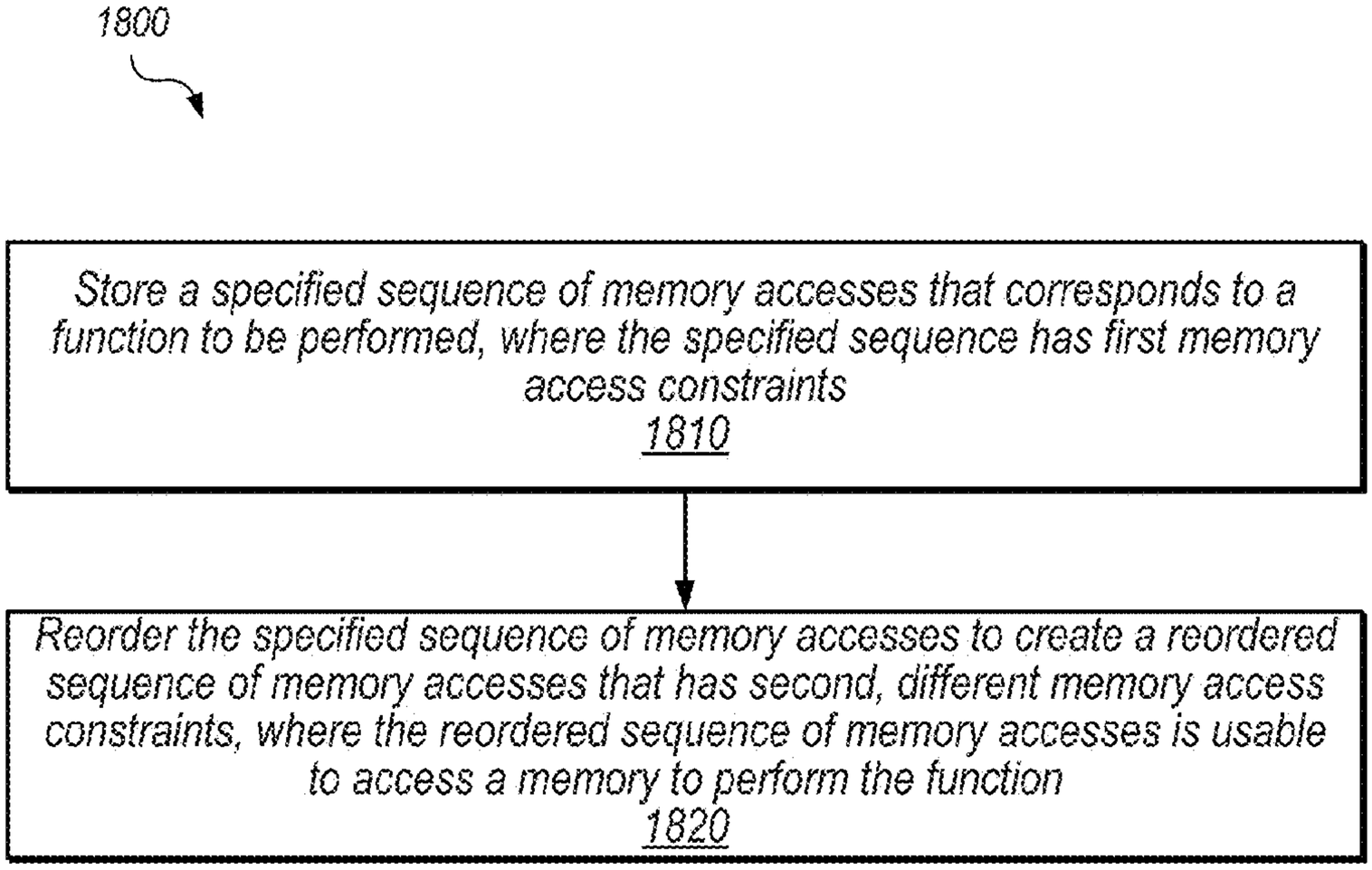

1800
Store a specified sequence of memory accesses that corresponds to a function to be performed, where the specified sequence has first memory access constraints
1810
Reorder the specified sequence of memory accesses to create a reordered sequence of memory accesses that has second, different memory access constraints, where the reordered sequence of memory accesses is usable to access a memory to perform the function
1820

Figure 18

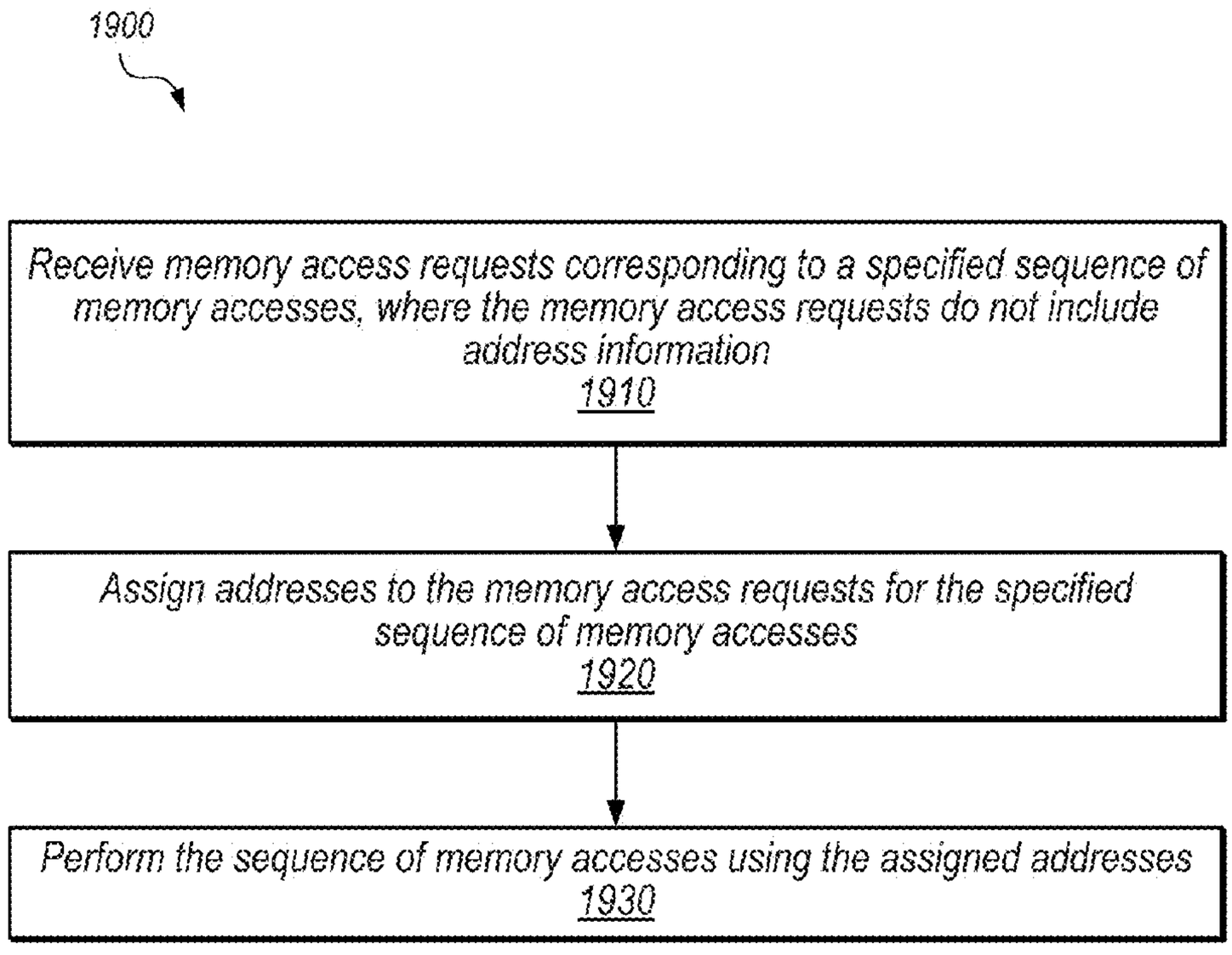

1900
Receive memory access requests corresponding to a specified sequence of memory accesses, where the memory access requests do not include address information
1910
Assign addresses to the memory access requests for the specified sequence of memory accesses
1920
Perform the sequence of memory accesses using the assigned addresses
1930

Figure 19

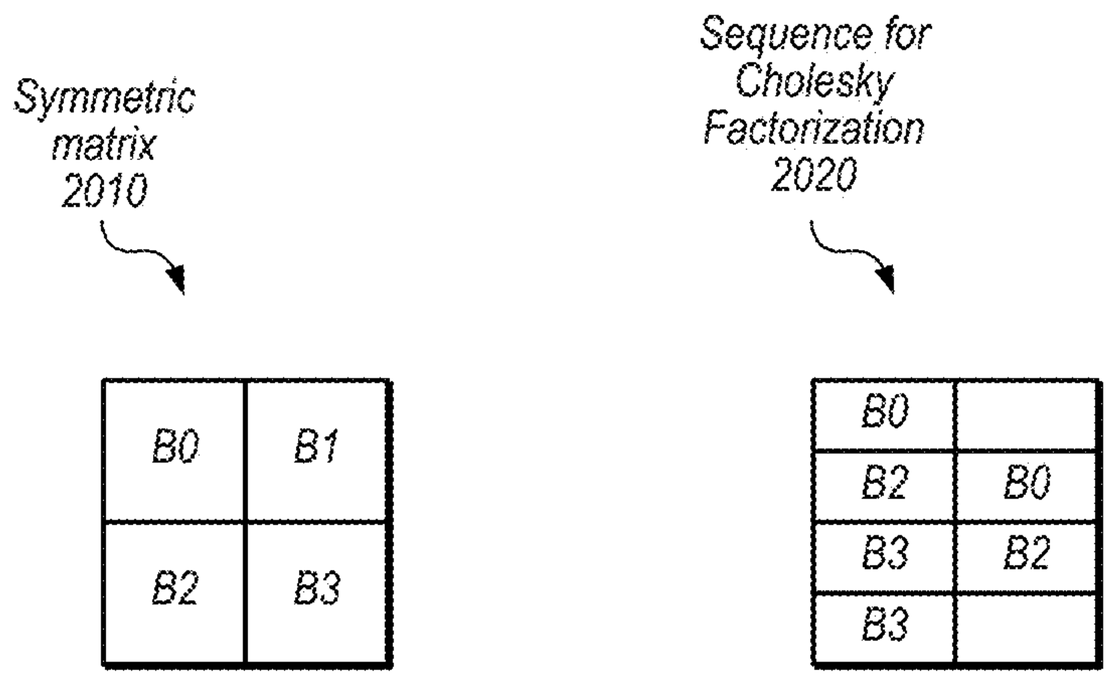

Symmetric matrix 2010
Sequence for Cholesky Factorization 2020
B0
B1
B2
B3
B0
B2
B0
B3
B2
B3

Figure 20

REORDERING A SEQUENCE OF MEMORY ACCESSES TO IMPROVE PIPELINED PERFORMANCE

PRIORITY DATA

[0001] This application claims the benefit of U.S. Provisional Application No. 62/045,832, titled "Memory Configured to Avoid Access Hazards," filed on Sep. 4, 2014, whose inventors are Tai A. Ly, Swapnil D. Mhaske, Hojin Kee, Adam T. Arnesen, David C. Uliana, and Newton G. Petersen, which is incorporated by reference herein in its entirety as though fully and completely set forth herein.

TECHNICAL FIELD

[0002] The present disclosure relates to the field of computer memory, and more particularly to techniques involving a memory system configured to resolve or prevent memory access hazards.

DESCRIPTION OF THE RELATED ART

[0003] Various computer programs require handling of hazards when accessing memory. Generally, when two operations access the same memory location, a computing system should ensure that the accesses occur in the proper order. Various algorithms use sequences of memory operations that are often performed iteratively. For example, error control codes, encryption, and various equation solving algorithms typically use complex matrices for which memory accesses must be performed in specified sequences for various operations.

[0004] Low-density parity-check (LDPC) codes are one example of error control code and are used to transmit information over potentially noisy channels. For example, the WiMAX and LTE standards utilize LDPC codes for data channels. LDPC typically uses a parity check matrix referred to as an "H matrix" to determine the makeup of incoming blocks of code. Decoding LDPC encoded messages involves iterative read/modify/write operations corresponding to different layers of the H matrix. Pipelining these operations may be desirable in order to decrease decode time, but dependencies between the memory accesses must be resolved to properly order the pipelined operations.

SUMMARY

[0005] Various embodiments of a memory configured to resolve or prevent access hazards, and techniques for configuring or using such a memory, are presented below.

[0006] In one embodiment, an apparatus includes a memory and circuitry coupled to or comprised in the memory. In this embodiment, the circuitry is configured to receive a sequence of memory access requests for the memory, where the sequence of memory access requests is configured to access locations associated with entries in a matrix. In this embodiment, the circuitry is configured with memory access constraints for the sequence of memory access requests. In this embodiment, the circuitry is configured to grant the sequence of memory access requests subject to the memory access constraints, thereby avoiding memory access hazards for a sequence of memory accesses corresponding to the sequence of memory access requests.

[0007] In some embodiments, the apparatus includes pipeline circuitry configured to pipeline performance of the sequence of memory accesses. In these embodiments, the circuitry may be configured to stall pipelined performance of the sequence of memory access requests based on the memory access constraints. The memory access constraints may be specified using any of various formats and encodings. For example, in some embodiments interlock information indicates read/write pointer values. In other embodiments, interlock information indicates token values. In some embodiments, the sequence of memory access requests is based on iterative operations for processing rows of a matrix. In some embodiments, the circuitry is configured to prevent memory access hazards without receiving other information indicating the memory access hazards, i.e., without auxiliary information describing or characterizing the memory access hazards.

[0008] In one embodiment, a method includes determining a sequence of memory access requests for a program and generating information specifying memory access constraints based on the sequence of memory accesses, where the information is usable to avoid memory access hazards for the sequence of memory accesses. In this embodiment, the method further includes configuring first circuitry using the information, where the first circuitry is included in or coupled to a memory. In this embodiment, after the configuring, the first circuitry is operable to perform memory access requests to the memory corresponding to the sequence of memory accesses while avoiding the memory access hazards, without receiving other information indicating the memory access hazards. The configuring the first circuitry may be performed using a programmable hardware element, such as a field programmable gate array (FPGA), or other programmable hardware. Furthermore, the configuring may be performed at compilation time, at start of running the program, or at various times while the program is running to adapt to changing conditions or to improve any number of performance measures such as throughput, latency, power consumption, reliability, resource utilization, bit-error rate, etc. In some embodiments, the determining is performed using compile-time execution.

[0009] In one embodiment, a method includes storing a specified sequence of memory accesses that corresponds to a function to be performed. In this embodiment, the specified sequence of memory accesses has first memory access constraints. In this embodiment, the method further includes reordering the specified sequence of memory accesses to create a reordered sequence of memory accesses that has second, different memory access constraints. In this embodiment, the reordered sequence of memory accesses is usable to access a memory to perform the function. In some embodiments, performance estimates are determined for a plurality of reordered sequences of memory accesses, and one of the reordered sequences is selected based on the performance estimates. In some embodiments, the reordered sequence is used to compile a program usable to perform the function.

[0010] In one embodiment, an apparatus includes a memory and addressing circuitry coupled to or comprised in the memory. In this embodiment, the addressing circuitry is configured to receive memory access requests corresponding to a specified sequence of memory accesses. In this embodiment, the memory access requests do not include address information. In this embodiment, the addressing circuitry is further configured to assign addresses to the memory access requests for the specified sequence of memory accesses. In some embodiments, the apparatus is configured to perform the memory access requests using the assigned addresses.

[0011] Various techniques disclosed herein may improve memory access throughput, reduce complexity in devices coupled to an interlock memory, and/or allow for flexible programming or updating of an interlock memory for different applications, in some embodiments.

BRIEF DESCRIPTION OF THE DRAWINGS

[0012] A better understanding of the present disclosure can be obtained when the following detailed description is considered in conjunction with the following drawings, in which:

[0013] FIG. **1**A illustrates a computer system configured to execute a program according to an embodiment of the present disclosure;

[0014] FIG. **1**B illustrates a network system comprising two or more computer systems that may implement an embodiment of the present disclosure;

[0015] FIG. **2**A illustrates an instrumentation control system according to one embodiment of the disclosure;

[0016] FIG. **2**B illustrates an industrial automation system according to one embodiment of the disclosure;

[0017] FIG. **3**A is a high level block diagram of an exemplary system which may execute or utilize programs including graphical programs;

[0018] FIG. **3**B illustrates an exemplary system which may perform control and/or simulation functions;

[0019] FIG. **4** is an exemplary block diagram of the computer systems of FIGS. **1**A, **1**B, **2**A and **2**B and **3**B;

[0020] FIG. **5** illustrates one embodiment of a system that includes a decoder with an interlock memory;

[0021] FIG. **6**A illustrates an exemplary H matrix;

[0022] FIG. **6**B illustrates one embodiment of a beta matrix and FIG. **6**C illustrates one embodiment of a rearranged beta matrix;

[0023] FIGS. **7**A and **7**B illustrate exemplary baseline decoding and pipelined decoding timelines respectively;

[0024] FIGS. **8**A and **8**B illustrate respective embodiments of an interlock memory;

[0025] FIG. **9** illustrates one embodiment of an interlock table;

[0026] FIG. **10** illustrates one embodiment of a method for operating an interlock memory;

[0027] FIG. **11** illustrates one embodiment of a method for generating information for an interlock memory using compile-time-execution;

[0028] FIG. **12** illustrates one embodiment of a method for determining performance estimates and rearranging memory access sequences based on performance estimates;

[0029] FIG. **13** illustrates one embodiment of a method for configuring an interlock memory;

[0030] FIG. **14** is a block diagram illustrating another embodiment of an interlock memory;

[0031] FIGS. **15**A and **15**B illustrate exemplary beta matrices, interlock tables, and pipelined execution of memory access operations according to one embodiment;

[0032] FIGS. **16**A and **16**B illustrate another embodiment of a method for operating an interlock memory;

[0033] FIG. **17** illustrates one embodiment of a method performed using an interlock memory;

[0034] FIG. **18** illustrates one embodiment of a method for reordering sequences of memory accesses;

[0035] FIG. **19** illustrates one embodiment of a method performed using a self-addressing memory; and

[0036] FIG. **20** illustrates a sequence of memory accesses for Cholesky factorization of a symmetric matrix.

[0037] While the disclosure is susceptible to various modifications and alternative forms, specific embodiments thereof are shown by way of example in the drawings and are herein described in detail. It should be understood, however, that the drawings and detailed description thereto are not intended to limit the invention to the particular form disclosed, but on the contrary, the intention is to cover all modifications, equivalents and alternatives falling within the spirit and scope of the present disclosure as defined by the appended claims.

[0038] The term "configured to" is used herein to connote structure by indicating that the units/circuits/components include structure (e.g., circuitry) that performs the task or tasks during operation. As such, the unit/circuit/component can be said to be configured to perform the task even when the specified unit/circuit/component is not currently operational (e.g., is not on). The units/circuits/components used with the "configured to" language include hardware—for example, circuits, memory storing program instructions executable to implement the operation, etc. Reciting that a unit/circuit/component is "configured to" perform one or more tasks is expressly intended not to invoke 35 U.S.C. §112(f) for that unit/circuit/component.

DETAILED DESCRIPTION

Incorporation by Reference

[0039] The following references are hereby incorporated by reference in their entirety as though fully and completely set forth herein:

[0040] U.S. Pat. No. 4,914,568 titled "Graphical System for Modeling a Process and Associated Method," issued on Apr. 3, 1990.

[0041] U.S. Pat. No. 5,481,741 titled "Method and Apparatus for Providing Attribute Nodes in a Graphical Data Flow Environment".

[0042] U.S. Pat. No. 6,173,438 titled "Embedded Graphical Programming System" filed Aug. 18, 1997.

[0043] U.S. Pat. No. 6,219,628 titled "System and Method for Configuring an Instrument to Perform Measurement Functions Utilizing Conversion of Graphical Programs into Hardware Implementations," filed Aug. 18, 1997.

[0044] U.S. Pat. No. 7,210,117 titled "System and Method for Programmatically Generating a Graphical Program in Response to Program Information," filed Dec. 20, 2000.

[0045] U.S. patent application Ser. No. 13/592,995 titled "Compile Time Execution," filed Aug. 23, 2012.

TERMS

[0046] The following is a glossary of terms used in the present application:

[0047] Memory Medium—Any of various types of non-transitory computer accessible memory devices or storage devices. The term "memory medium" is intended to include an installation medium, e.g., a CD-ROM, floppy disks **104**, or tape device; a computer system memory or random access memory such as DRAM, DDR RAM, SRAM, EDO RAM, Rambus RAM, etc.; a non-volatile memory such as a Flash, magnetic media, e.g., a hard drive, or optical storage; registers, or other similar types of memory elements, etc. The memory medium may comprise other types of non-transitory memory as well or combinations thereof. In addition, the memory medium may be located in a first computer in which the programs are executed, or may be located in a second

different computer which connects to the first computer over a network, such as the Internet. In the latter instance, the second computer may provide program instructions to the first computer for execution. The term "memory medium" may include two or more memory mediums which may reside in different locations, e.g., in different computers that are connected over a network.

[0048] Carrier Medium—a memory medium as described above, as well as a physical transmission medium, such as a bus, network, and/or other physical transmission medium that conveys signals such as electrical, electromagnetic, or digital signals.

[0049] Programmable Hardware Element—includes various hardware devices comprising multiple programmable function blocks connected via a programmable interconnect. Examples include FPGAs (Field Programmable Gate Arrays), PLDs (Programmable Logic Devices), FPOAs (Field Programmable Object Arrays), and CPLDs (Complex PLDs). The programmable function blocks may range from fine grained (combinatorial logic or look up tables) to coarse grained (arithmetic logic units or processor cores). A programmable hardware element may also be referred to as "reconfigurable logic".

[0050] Software Program—the term "software program" is intended to have the full breadth of its ordinary meaning, and includes any type of program instructions, code, script and/or data, or combinations thereof, that may be stored in a memory medium and executed by a processor. Exemplary software programs include programs written in text-based programming languages, such as C, C++, PASCAL, FORTRAN, COBOL, JAVA, assembly language, etc.; graphical programs (programs written in graphical programming languages); assembly language programs; programs that have been compiled to machine language; scripts; and other types of executable software. A software program may comprise two or more software programs that interoperate in some manner. Note that various embodiments described herein may be implemented by a computer or software program. A software program may be stored as program instructions on a memory medium.

[0051] Hardware Configuration Program—a program, e.g., a netlist or bit file, that can be used to program or configure a programmable hardware element.

[0052] Program—the term "program" is intended to have the full breadth of its ordinary meaning. The term "program" includes 1) a software program which may be stored in a memory and is executable by a processor or 2) a hardware configuration program useable for configuring a programmable hardware element.

[0053] Graphical Program—A program comprising a plurality of interconnected nodes or icons, wherein the plurality of interconnected nodes or icons visually indicate functionality of the program. The interconnected nodes or icons are graphical source code for the program. Graphical function nodes may also be referred to as blocks.

[0054] The following provides examples of various aspects of graphical programs. The following examples and discussion are not intended to limit the above definition of graphical program, but rather provide examples of what the term "graphical program" encompasses:

[0055] The nodes in a graphical program may be connected in one or more of a data flow, control flow, and/or execution flow format. The nodes may also be connected in a "signal flow" format, which is a subset of data flow.

[0056] Exemplary graphical program development environments which may be used to create graphical programs include LabVIEW®, DasyLab™, DIADem™ and Matrixx/SystemBuild™ from National Instruments, Simulink® from the MathWorks, VEE™ from Agilent, WiT™ from Coreco, Vision Program Manager™ from PPT Vision, SoftWIRE™ from Measurement Computing, Sanscript™ from Northwoods Software, Khoros™ from Khoral Research, SnapMaster™ from HEM Data, VisSim™ from Visual Solutions, ObjectBench™ by SES (Scientific and Engineering Software), and VisiDAQ™ from Advantech, among others.

[0057] The term "graphical program" includes models or block diagrams created in graphical modeling environments, wherein the model or block diagram comprises interconnected blocks (i.e., nodes) or icons that visually indicate operation of the model or block diagram; exemplary graphical modeling environments include Simulink®, SystemBuild™, VisSim™, Hypersignal Block Diagram™, etc.

[0058] A graphical program may be represented in the memory of the computer system as data structures and/or program instructions. The graphical program, e.g., these data structures and/or program instructions, may be compiled or interpreted to produce machine language that accomplishes the desired method or process as shown in the graphical program.

[0059] Input data to a graphical program may be received from any of various sources, such as from a device, unit under test, a process being measured or controlled, another computer program, a database, or from a file. Also, a user may input data to a graphical program or virtual instrument using a graphical user interface, e.g., a front panel.

[0060] A graphical program may optionally have a GUI associated with the graphical program. In this case, the plurality of interconnected blocks or nodes are often referred to as the block diagram portion of the graphical program.

[0061] Node—In the context of a graphical program, an element that may be included in a graphical program. The graphical program nodes (or simply nodes) in a graphical program may also be referred to as blocks. A node may have an associated icon that represents the node in the graphical program, as well as underlying code and/or data that implements functionality of the node. Exemplary nodes (or blocks) include function nodes, sub-program nodes, terminal nodes, structure nodes, etc. Nodes may be connected together in a graphical program by connection icons or wires.

[0062] Data Flow Program—A Software Program in which the program architecture is that of a directed graph specifying the flow of data through the program, and thus functions execute whenever the necessary input data are available. Said another way, data flow programs execute according to a data flow model of computation under which program functions are scheduled for execution in response to their necessary input data becoming available. Data flow programs can be contrasted with procedural programs, which specify an execution flow of computations to be performed. As used herein "data flow" or "data flow programs" refer to "dynamically-scheduled data flow" and/or "statically-defined data flow".

[0063] Graphical Data Flow Program (or Graphical Data Flow Diagram)—A Graphical Program which is also a Data Flow Program. A Graphical Data Flow Program comprises a plurality of interconnected nodes (blocks), wherein at least a subset of the connections among the nodes visually indicate that data produced by one node is used by another node. A

LabVIEW VI is one example of a graphical data flow program. A Simulink block diagram is another example of a graphical data flow program.

[0064] Graphical User Interface—this term is intended to have the full breadth of its ordinary meaning. The term "Graphical User Interface" is often abbreviated to "GUI". A GUI may comprise only one or more input GUI elements, only one or more output GUI elements, or both input and output GUI elements.

[0065] The following provides examples of various aspects of GUIs. The following examples and discussion are not intended to limit the ordinary meaning of GUI, but rather provide examples of what the term "graphical user interface" encompasses:

[0066] A GUI may comprise a single window having one or more GUI Elements, or may comprise a plurality of individual GUI Elements (or individual windows each having one or more GUI Elements), wherein the individual GUI Elements or windows may optionally be tiled together.

[0067] A GUI may be associated with a graphical program. In this instance, various mechanisms may be used to connect GUI Elements in the GUI with nodes in the graphical program. For example, when Input Controls and Output Indicators are created in the GUI, corresponding nodes (e.g., terminals) may be automatically created in the graphical program or block diagram. Alternatively, the user can place terminal nodes in the block diagram which may cause the display of corresponding GUI Elements front panel objects in the GUI, either at edit time or later at run time. As another example, the GUI may comprise GUI Elements embedded in the block diagram portion of the graphical program.

[0068] Front Panel—A Graphical User Interface that includes input controls and output indicators, and which enables a user to interactively control or manipulate the input being provided to a program, and view output of the program, while the program is executing.

[0069] A front panel is a type of GUI. A front panel may be associated with a graphical program as described above.

[0070] In an instrumentation application, the front panel can be analogized to the front panel of an instrument. In an industrial automation application the front panel can be analogized to the MMI (Man Machine Interface) of a device. The user may adjust the controls on the front panel to affect the input and view the output on the respective indicators.

[0071] Graphical User Interface Element—an element of a graphical user interface, such as for providing input or displaying output. Exemplary graphical user interface elements comprise input controls and output indicators.

[0072] Input Control—a graphical user interface element for providing user input to a program. An input control displays the value input by the user and is capable of being manipulated at the discretion of the user. Exemplary input controls comprise dials, knobs, sliders, input text boxes, etc.

[0073] Output Indicator—a graphical user interface element for displaying output from a program. Exemplary output indicators include charts, graphs, gauges, output text boxes, numeric displays, etc. An output indicator is sometimes referred to as an "output control".

[0074] Computer System—any of various types of computing or processing systems, including a personal computer system (PC), mainframe computer system, workstation, network appliance, Internet appliance, personal digital assistant (PDA), television system, grid computing system, or other device or combinations of devices. In general, the term "computer system" can be broadly defined to encompass any device (or combination of devices) having at least one processor that executes instructions from a memory medium.

[0075] Measurement Device—includes instruments, data acquisition devices, smart sensors, and any of various types of devices that are configured to acquire and/or store data. A measurement device may also optionally be further configured to analyze or process the acquired or stored data. Examples of a measurement device include an instrument, such as a traditional stand-alone "box" instrument, a computer-based instrument (instrument on a card) or external instrument, a data acquisition card, a device external to a computer that operates similarly to a data acquisition card, a smart sensor, one or more DAQ or measurement cards or modules in a chassis, an image acquisition device, such as an image acquisition (or machine vision) card (also called a video capture board) or smart camera, a motion control device, a robot having machine vision, and other similar types of devices. Exemplary "stand-alone" instruments include oscilloscopes, multimeters, signal analyzers, arbitrary waveform generators, spectroscopes, and similar measurement, test, or automation instruments.

[0076] A measurement device may be further configured to perform control functions, e.g., in response to analysis of the acquired or stored data. For example, the measurement device may send a control signal to an external system, such as a motion control system or to a sensor, in response to particular data. A measurement device may also be configured to perform automation functions, i.e., may receive and analyze data, and issue automation control signals in response.

[0077] Functional Unit (or Processing Element)—refers to various elements or combinations of elements. Processing elements include, for example, circuits such as an ASIC (Application Specific Integrated Circuit), portions or circuits of individual processor cores, entire processor cores, individual processors, programmable hardware devices such as a field programmable gate array (FPGA), and/or larger portions of systems that include multiple processors, as well as any combinations thereof.

[0078] Automatically—refers to an action or operation performed by a computer system (e.g., software executed by the computer system) or device (e.g., circuitry, programmable hardware elements, ASICs, etc.), without user input directly specifying or performing the action or operation. Thus the term "automatically" is in contrast to an operation being manually performed or specified by the user, where the user provides input to directly perform the operation. An automatic procedure may be initiated by input provided by the user, but the subsequent actions that are performed "automatically" are not specified by the user, i.e., are not performed "manually", where the user specifies each action to perform. For example, a user filling out an electronic form by selecting each field and providing input specifying information (e.g., by typing information, selecting check boxes, radio selections, etc.) is filling out the form manually, even though the computer system must update the form in response to the user actions. The form may be automatically filled out by the computer system where the computer system (e.g., software executing on the computer system) analyzes the fields of the form and fills in the form without any user input specifying the answers to the fields. As indicated above, the user may invoke the automatic filling of the form, but is not involved in the actual filling of the form (e.g., the user is not manually specifying answers to fields but rather they are being auto-

matically completed). The present specification provides various examples of operations being automatically performed in response to actions the user has taken.

[0079] Concurrent—refers to parallel execution or performance, where tasks, processes, or programs are performed in an at least partially overlapping manner. For example, concurrency may be implemented using "strong" or strict parallelism, where tasks are performed (at least partially) in parallel on respective computational elements, or using "weak parallelism", where the tasks are performed in an interleaved manner, e.g., by time multiplexing of execution threads.

FIG. **1**A—Computer System

[0080] FIG. **1**A illustrates an exemplary computer system **82** configured to execute a program, e.g., a graphical program, configured to program and/or utilize an interlock memory, according to some embodiments. Note that while some of the embodiments are described in terms of a graphical program, the techniques disclosed are broadly applicable to text based programs, as well, and so the described graphical program aspects of the described embodiments should be considered to be exemplary only, i.e., the novel techniques described with respect to graphical programs also apply to text based programs.

[0081] As shown in FIG. **1**A, the computer system **82** may include a display device configured to display a graphical program as the graphical program is created and/or executed. The display device may also be configured to display a graphical user interface or front panel of the graphical program during execution of the graphical program. The graphical user interface may comprise any type of graphical user interface, e.g., depending on the computing platform.

[0082] The computer system **82** may include at least one memory medium on which one or more computer programs or software components according to one embodiment of the present disclosure may be stored. For example, the memory medium may store one or more graphical (or text based) programs which are executable to perform the methods described herein. Additionally, the memory medium may store a graphical (or text based) programming development environment application used to create and/or execute such graphical (or text based) programs. The memory medium may also store operating system software, as well as other software for operation of the computer system. Various embodiments further include receiving or storing instructions and/or data implemented in accordance with the foregoing description upon a carrier medium.

FIG. **1**B—Computer Network

[0083] FIG. **1**B illustrates a system including a first computer system **82** that is coupled to a second computer system **90**. The computer system **82** may be coupled via a network **84** (or a computer bus) to the second computer system **90**. The computer systems **82** and **90** may each be any of various types, as desired. The network **84** can also be any of various types, including a LAN (local area network), WAN (wide area network), the Internet, or an Intranet, among others. The computer systems **82** and **90** may execute a graphical program in a distributed fashion. For example, computer **82** may execute a first portion of the block diagram of a graphical program and computer system **90** may execute a second portion of the block diagram of the graphical program. As another example, computer **82** may display the graphical user interface of a graphical program and computer system **90** may execute the block diagram of the graphical program.

[0084] In one embodiment, the graphical user interface of the graphical program may be displayed on a display device of the computer system **82**, and the block diagram may execute on a device coupled to the computer system **82**. The device may include a programmable hardware element and/or may include a processor and memory medium which may execute a real time operating system. In one embodiment, the graphical program may be downloaded and executed on the device. For example, an application development environment with which the graphical program is associated may provide support for downloading a graphical program for execution on the device in a real time system.

Exemplary Systems

[0085] Embodiments of the present disclosure may be involved with performing test and/or measurement functions; controlling and/or modeling instrumentation or industrial automation hardware; modeling and simulation functions, e.g., modeling or simulating a device or product being developed or tested, etc. Exemplary test applications where the graphical program may be used include hardware-in-the-loop testing and rapid control prototyping, among others.

[0086] However, it is noted that embodiments of the present invention can be used for a plethora of applications and is not limited to the above applications. In other words, applications discussed in the present description are exemplary only, and embodiments of the present invention may be used in any of various types of systems. Thus, embodiments of the system and method of the present invention is configured to be used in any of various types of applications, including the control of other types of devices such as multimedia devices, video devices, audio devices, telephony devices, Internet devices, etc., as well as general purpose software applications such as word processing, spreadsheets, network control, network monitoring, financial applications, games, etc.

[0087] FIG. **2**A illustrates an exemplary instrumentation control system **100** which may implement embodiments of the invention. The system **100** comprises a host computer **82** which couples to one or more instruments. The host computer **82** may comprise a CPU, a display screen, memory, and one or more input devices such as a mouse or keyboard as shown. The computer **82** may operate with the one or more instruments to analyze, measure or control a unit under test (UUT) or process **150**, e.g., via execution of software **104**.

[0088] The one or more instruments may include a GPIB instrument **112** and associated GPIB interface card **122**, a data acquisition board **114** inserted into or otherwise coupled with chassis **124** with associated signal conditioning circuitry **126**, a VXI instrument **116**, a PXI instrument **118**, a video device or camera **132** and associated image acquisition (or machine vision) card **134**, a motion control device **136** and associated motion control interface card **138**, and/or one or more computer based instrument cards **142**, among other types of devices. The computer system may couple to and operate with one or more of these instruments. The instruments may be coupled to the unit under test (UUT) or process **150**, or may be coupled to receive field signals, typically generated by transducers. The system **100** may be used in a data acquisition and control application, in a test and measurement application, an image processing or machine vision application, a process control application, a man-machine

interface application, a simulation application, or a hardware-in-the-loop validation application, among others.

[0089] FIG. **2**B illustrates an exemplary industrial automation system **200** which may implement embodiments of the invention. The industrial automation system **200** is similar to the instrumentation or test and measurement system **100** shown in FIG. **2**A. Elements which are similar or identical to elements in FIG. **2**A have the same reference numerals for convenience. The system **200** may comprise a computer **82** which couples to one or more devices or instruments. The computer **82** may comprise a CPU, a display screen, memory, and one or more input devices such as a mouse or keyboard as shown. The computer **82** may operate with the one or more devices to perform an automation function with respect to a process or device **150**, such as MMI (Man Machine Interface), SCADA (Supervisory Control and Data Acquisition), portable or distributed data acquisition, process control, advanced analysis, or other control, among others, e.g., via execution of software **104**.

[0090] The one or more devices may include a data acquisition board **114** inserted into or otherwise coupled with chassis **124** with associated signal conditioning circuitry **126**, a PXI instrument **118**, a video device **132** and associated image acquisition card **134**, a motion control device **136** and associated motion control interface card **138**, a fieldbus device **270** and associated fieldbus interface card **172**, a PLC (Programmable Logic Controller) **176**, a serial instrument **282** and associated serial interface card **184**, or a distributed data acquisition system, such as Fieldpoint system **185**, available from National Instruments Corporation, among other types of devices.

[0091] FIG. **3**A is a high level block diagram of an exemplary system which may execute or utilize programs, e.g., graphical programs. FIG. **3**A illustrates a general high-level block diagram of a generic control and/or simulation system which comprises a controller **92** and a plant **94**. The controller **92** represents a control system/algorithm the user may be trying to develop. The plant **94** represents the system the user may be trying to control. For example, if the user is designing an ECU for a car, the controller **92** is the ECU and the plant **94** is the car's engine (and possibly other components such as transmission, brakes, and so on.) As shown, a user may create a graphical program that specifies or implements the functionality of one or both of the controller **92** and the plant **94**. For example, a control engineer may use a modeling and simulation tool to create a model (graphical program) of the plant **94** and/or to create the algorithm (graphical program) for the controller **92**.

[0092] FIG. **3**B illustrates an exemplary system which may perform control and/or simulation functions. As shown, the controller **92** may be implemented by a computer system **82** or other device (e.g., including a processor and memory medium and/or including a programmable hardware element) that executes or implements a graphical program. In a similar manner, the plant **94** may be implemented by a computer system or other device **144** (e.g., including a processor and memory medium and/or including a programmable hardware element) that executes or implements a graphical program, or may be implemented in or as a real physical system, e.g., a car engine.

[0093] In one embodiment, one or more graphical programs may be created which are used in performing rapid control prototyping. Rapid Control Prototyping (RCP) generally refers to the process by which a user develops a control algorithm and quickly executes that algorithm on a target controller connected to a real system. The user may develop the control algorithm using a graphical program, and the graphical program may execute on the controller **92**, e.g., on a computer system or other device. The computer system **82** may be a platform that supports real time execution, e.g., a device including a processor that executes a real time operating system (RTOS), or a device including a programmable hardware element.

[0094] In one embodiment, one or more graphical programs may be created which are used in performing Hardware in the Loop (HIL) simulation. Hardware in the Loop (HIL) refers to the execution of the plant model **94** in real time to test operation of a real controller **92**. For example, once the controller **92** has been designed, it may be expensive and complicated to actually test the controller **92** thoroughly in a real plant, e.g., a real car. Thus, the plant model (implemented by a graphical program) is executed in real time to make the real controller **92** "believe" or operate as if it is connected to a real plant, e.g., a real engine.

[0095] In the embodiments of FIGS. **2**A, **2**B, and **3**B above, one or more of the various devices may couple to each other over a network, such as the Internet. In one embodiment, the user operates to select a target device from a plurality of possible target devices for programming or configuration using a program, e.g., a graphical program. Thus the user may create a graphical program on a computer and use (execute) the graphical program on that computer or deploy the graphical program to a target device (for remote execution on the target device) that is remotely located from the computer and coupled to the computer through a network.

[0096] Graphical software programs which perform data acquisition, analysis and/or presentation, e.g., for measurement, instrumentation control, industrial automation, modeling, or simulation, such as in the applications shown in FIGS. **2**A and **2**B, may be referred to as virtual instruments.

FIG. **4**—Computer System Block Diagram

[0097] FIG. **4** is a block diagram **12** representing one embodiment of the computer system **82** and/or **90** illustrated in FIGS. **1**A and **1**B, or computer system **82** shown in FIG. **2**A or **2**B. It is noted that any type of computer system configuration or architecture can be used as desired, and FIG. **4** illustrates a representative PC embodiment. It is also noted that the computer system may be a general purpose computer system, a computer implemented on a card installed in a chassis, or other types of embodiments. Elements of a computer not necessary to understand the present description have been omitted for simplicity.

[0098] The computer may include at least one central processing unit or CPU (processor) **160** which is coupled to a processor or host bus **162**. The CPU **160** may be any of various types, including an x86 processor, e.g., a Pentium class, a PowerPC processor, a CPU from the SPARC family of RISC processors, as well as others. A memory medium, typically comprising RAM and referred to as main memory, **166** is coupled to the host bus **162** by means of memory controller **164**. The main memory **166** may store the one or more programs configured to program and/or utilize an interlock memory. The main memory may also store operating system software, as well as other software for operation of the computer system.

[0099] The host bus **162** may be coupled to an expansion or input/output bus **170** by means of a bus controller **168** or bus

bridge logic. The expansion bus **170** may be the PCI (Peripheral Component Interconnect) expansion bus, although other bus types can be used. The expansion bus **170** includes slots for various devices such as described above. The computer **82** further comprises a video display subsystem **180** and hard drive **182** coupled to the expansion bus **170**. The computer **82** may also comprise a GPIB card **122** coupled to a GPIB bus **112**, and/or an MXI device **186** coupled to a VXI chassis **116**.

[0100] As shown, a device **190** may also be connected to the computer. The device **190** may include a processor and memory which may execute a real time operating system. The device **190** may also or instead comprise a programmable hardware element. The computer system may be configured to deploy a program, e.g., a graphical program to the device **190** for execution of the graphical program on the device **190**. The deployed graphical program may take the form of graphical program instructions or data structures that directly represents the graphical program. Alternatively, the deployed graphical program may take the form of text code (e.g., C code) generated from the graphical program. As another example, the deployed graphical program may take the form of compiled code generated from either the graphical program or from text code that in turn was generated from the graphical program. As noted above, in some embodiments, the program(s) may be text based.

Exemplary LDPC Decoder

[0101] Various embodiments discussed below with reference to FIGS. **5-16** involve techniques associated with decoding error correcting code, and low-density parity-check (LDPC) codes in particular. However, similar techniques may be applied to various algorithms in addition to and/or in place of such decoding. LDPC codes are discussed for explanatory purposes but are not intended to limit the scope of the present techniques. Embodiments of the interlock memory, performance modeling, hardware programming, memory sequence rearrangement, and/or self-addressing techniques discussed herein may be applied in various contexts for any of various algorithms which may or may not be explicitly listed herein.

[0102] Examples of such algorithms include, without limitation: error control code, graph encoding/decoding, source coding, cryptography, maximum likelihood detector, maximum a posteriori detector, compression, multiple-input multiple-output (MIMO) communications, beam-forming, beam-steering, differential equation solving, linear equation solving, linear algebra, optimization, detection and estimation, networking, machine learning, channel estimation, image processing, motion control, process control, bioinformatics, dynamic programming, big data applications, computational informatics, internet of things, etc. Thus sequences of memory accesses may be based on, and/or derived from, one or more of, without limitation: a parity-check matrix, a generator matrix, a channel matrix, the number of nodes and/or edges in a decoding algorithm, number of antennae, number of channel taps, compression ratio, angle range, degree of polynomial, number of equations, local and/or global maxima and/or minima, number of network nodes, etc.

[0103] Exemplary embodiments of linear algebra algorithms include, without limitation: symmetric rank-k update, symmetric rank-2k update, Cholesky factorization (decomposition of a positive-definite matrix into a product of a lower triangular matrix and its conjugate transpose), update Cholesky factorization (e.g., when a previously-decomposed matrix changes in some way), lower upper (LU) factorization (decomposition of a matrix into a product of a lower triangular matrix and an upper triangular matrix), QR factorization (decomposition of a matrix into a product of an orthogonal matrix Q and an upper triangular matrix R), update QR factorization, LQ factorization, triangular matrix inversion, reduction to Hessenberg form, reduction to bi-diagonal form, reduction to tri-diagonal form, triangular matrix Lyapunov equation solutions, triangular matrix Sylvester equation solutions, etc.

[0104] FIG. **5** shows one embodiment of an exemplary system that includes an LDPC decoder. In the illustrated embodiment, system **500** includes encoder **510**, decoder **520**, modulator **530**, and demodulator **540**.

[0105] Encoder **510**, in the illustrated embodiment, is configured to receive a message to be transmitted, encode the message, e.g., using an LDPC generator matrix, and send the encoded message to modulator **530**.

[0106] Modulator **530**, in the illustrated embodiment, is configured to transform and/or group bits of the encoded message into symbols for wireless transmission to an antenna coupled to demodulator **540**, i.e., to modulate the encoded message for transmission. The wireless transmission may conform to any of various transmission standards. In other embodiments, the transmission may not be wireless.

[0107] Demodulator **540**, in the illustrated embodiment, is configured to demodulate the received signals to generate the encoded message. However, because of noise in transmission, it may be difficult to determine the value of elements (e.g., the polarity of bits) in the encoded message. LDPC decoding may allow accurate reconstruction of the original message in many situations.

[0108] Decoder **520**, in the illustrated embodiment, is configured to reconstruct the original message, e.g., based on a parity check matrix (which is often denoted using the letter 'H' and may be referred to as an "H matrix"), i.e., to decode the encoded message, thereby recovering the original message. In the illustrated embodiment, decoder **520** includes interlock memory **550**, which may be configured to resolve or prevent access hazards corresponding to operations associated with the parity check matrix. Embodiments of interlock memory **550** are described in further detail below. Decoder **520** may include a pipelined data path for performing memory accesses based on the parity check matrix, and various techniques disclosed herein may reduce latency in the pipelined data path. Decoder **520** may be configured to transmit the decoded message to recipient hardware, not shown.

[0109] FIG. **6**A shows a diagram illustrating an exemplary embodiment of an H matrix **620**. In the illustrated embodiment, H matrix **620** is a parity check matrix and the number in each position indicates a cyclic shift amount. The blank positions in H matrix **620** indicate a "don't care," where the shift amount is not indicated, e.g., because it is not used.

[0110] In some embodiments, to decode a received signal using LDPC, decoder **520** is configured to iterate through the layers of H matrix **620** (12 layers in the illustrated embodiment, which correspond to the rows of the matrix) to decode a received signal. For example, decoder **520** may read the relevant positions in a layer, determine a minimum or maximum value, and update values for the layer based on the determined minimum or maximum before proceeding to the next layer. In the illustrated embodiment, H matrix **620** is a 12 by 24 matrix, but any of various matrix sizes may be used in other embodiments.

[0111] FIG. **6**B shows a diagram illustrating an exemplary embodiment of a beta matrix **640**. Beta matrix **640**, in the illustrated embodiment, encodes memory read/write addresses for iterative processing of layers L1 through L12 of H matrix **620**. The encoded addresses correspond to locations in interlock memory **550** in some embodiments. In the illustrated embodiment, beta matrix **640** is formed based on H matrix **620**, and indicates the block index of positions in H matrix **620** that are relevant (e.g., are not "don't cares"). For example, the position at layer L1 and block index 4 in H matrix **620** has a value of 50, so that block index (4) is included in layer L1 in beta matrix **640**. In the illustrated embodiment, beta matrix **640** also includes some blank "don't care" positions, which may be included in order to maintain a regular matrix.

[0112] In the illustrated embodiment, beta matrix **640** includes only a third the number of columns included in H matrix **620**. This may increase LDPC decoding performance significantly, e.g., by up to three times in some embodiments, relative to performing operations for each position in H matrix **620**.

[0113] Decoder **520**, in some embodiments, is configured to process beta matrix **640** iteratively in layers. In this example, to process the first layer, decoder **520** is configured to read memory locations 0, 4, 6, 8, 10, 12, and 13, modify at least some of the retrieved data, and write modified data back to at least some of the same locations. In one embodiment, decoder **520** is configured to calculate a minimum value for a given layer and calculate new values to write back to the same locations based on the minimum value, before proceeding to the next layer. Thus, operations for a given layer may depend on the values written for the previous layer. It may be desirable to pipeline memory accesses to improve performance. However, note for example that the first location in layer 2 is the same as the first location in layer 1 (both location 0). Thus, the read for layer 2 for location 0 is dependent on the write for layer 1. In some embodiments, interlock memory **550** is configured to resolve this dependency by waiting to perform the read for layer 2 until the write for layer 1 has completed. This stalling of the read may ensure correctness. However, reducing such stalling may be desirable in order to increase read/write performance in pipelined implementations.

[0114] FIG. **6**C shows a diagram illustrating one embodiment of a rearranged beta matrix **660**. In the illustrated embodiment, rearranged beta matrix is formed by rearranging the positions in each row of beta matrix **640**. This rearrangement may reduce read/write hazards and thus improve memory access performance. For example, consider the accesses to location 0 shown in bold. The read to location 0 for layer two has been shifted such that the write to location 0 for layer one may be completed before the read for layer two, thus avoiding the need to stall in pipelined implementations. Thus, in various embodiments, sequences of memory accesses may be rearranged to improve memory performance. This rearranging may be performed automatically by a compiler and/or by configuration circuitry for interlock memory **550**, for example. Techniques for performance modeling and selecting a rearranged sequence of memory accesses are discussed below with reference to FIG. **12**.

[0115] Referring now to FIG. **7**A, a diagram illustrating a timeline of baseline decoding **710** according to one embodiment is shown. In FIGS. **7**A-**7**B, shaded rectangles indicate writes while non-shaded rectangles indicate reads.

[0116] In the example of FIG. **7**A, reads occur for all locations for layer 1 of beta matrix **660**, which takes at least eight cycles (assuming that one read is initiated per cycle). All locations for layer 1 are then written, taking at least another eight cycles in the illustrated example. Processing for layer two then begins with a similar sequence of reads followed by a sequence of writes.

[0117] Referring now to FIG. **7**B, a diagram illustrating a timeline of pipelined decoding **720** according to one embodiment is shown. In some embodiments, decoder **520** is configured to pipeline performance of memory accesses. In the illustrated example, reads for layer 2 begin as soon as reads for layer 1 are finished. This may improve decoding performance substantially, e.g., by up to two times, relative to the baseline decoding **710**. Note that hazards must still be resolved or prevented (e.g., by inserting bubbles into the pipeline), but that rearranging a sequence of memory accesses (e.g., using rearranged beta matrix **660**) may greatly reduce hazards in some embodiments.

[0118] Various pipelined memory access approaches are discussed herein, but it should be noted that in other embodiments the disclosed techniques may be used in the context of non-pipelined memory accesses.

[0119] In some embodiments, memory accesses may be fully pipelined such that one or more reads and writes are issued every cycle if available, barring stalls to resolve hazards (not shown in FIG. **7**B). In some embodiments, a memory may be multi-ported, allowing multiple reads and/or writes to issue in a given cycle, which may further increase performance relative to baseline decoding **710**.

Exemplary Interlock Memory

[0120] FIG. **8**A shows a block diagram illustrating one embodiment of interlock memory **550**. In the illustrated embodiment, interlock memory **550** includes memory **810** and arbiter **830**. Arbiter **830**, in the illustrated embodiment, includes circuitry configured to receive read and write requests and grant the requests, resolving or preventing/avoiding any hazards in accessing memory **810** in the process. Interlock memory **550** may be configured to avoid memory access hazards without receiving information indicating the hazards along with memory access requests corresponding to the sequence (e.g., interlock memory **550** may be configured to avoid the hazards based on internal configuration). Arbiter **830** may be configured with memory access constraints for the sequence of memory access requests, exemplary embodiments of which are described below with reference to FIGS. **9** and **14**-**15**. These memory access constrains may be referred to as "interlock information." In some embodiments, interlock information is populated based on a specified sequence of memory accesses to be performed using interlock memory **550**.

[0121] As used herein, circuitry being "configured with" or "configured using" memory access constraints is intended to include configurations in which: 1) the circuitry stores information, e.g., such as a data structure or table which contains information regarding the memory access constraints 2) the circuitry includes dedicated circuitry configured to implement the memory access constraints (e.g., an ASIC), and/or 3) the circuitry includes programmable circuitry that is programmed to implement the memory access constraints. Hazard circuitry configured with memory access constraints may

be grouped in one location or may be distributed (e.g., among stages of a pipelined memory access data path), in some embodiments.

[0122] Memory **810**, in the illustrated embodiment, is configured to store received write data and provide read data, as allowed by arbiter **830**. In some embodiments, memory **810** is multi-ported to allow initiation of multiples reads and/or multiple writes in a given cycle. For example, dual-ported embodiments of memory **810** may allow initiation of writes to two different locations in the same cycle. In some embodiments, memory **810** includes multiple banks. In some embodiments, each bank allows initiation of only one read and one write per cycle, but different banks may be accessed in the same cycle, e.g., concurrently.

[0123] FIG. **8**B shows a block diagram illustrating another embodiment of interlock memory **550**. In the illustrated embodiment, interlock memory **550** includes address generator **820**. Other elements of FIG. **8**B may be configured as described above with reference to similarly numbered elements of FIG. **8**A. Address generator **820**, in the illustrated embodiment, includes circuitry configured to generate addresses for read and write requests. For example, interlock memory **550** may be configured for a particular array or matrix of memory accesses and address generator **820** may be configured to assign addresses to requests based on the sequence in which they are received. This may allow interlock memory **550** to be self-addressing, in some embodiments, such that programs can simply provide memory access requests, without address information, and interlock memory **550** is configured to generate appropriate addresses for corresponding accesses. In some embodiments, the sequence and/or addresses of memory accesses in a program are determined during compilation, which may include compile-time execution as discussed below with reference to FIG. **11**. In some embodiments, interlock memory **550** includes one or more programmable hardware elements such that it is easily reconfigurable for different programs and/or algorithms.

[0124] In some embodiments, interlock memory **550** may include additional elements (not shown) configured to perform data forwarding, caching, prefetching, etc. associated with memory systems. In some embodiments, interlock memory **550** includes look-ahead circuitry configured to perform and/or adjust one or more of caching, prefetching, data forwarding, etc., based on upcoming memory access addresses to improve performance. In various embodiments, interlock memory **550** is included in a device such as an LDPC decoder or any of various appropriate devices. The various techniques discussed herein may be used to improve performance of any of various devices that include one or more interlock memories.

[0125] FIG. **9** shows a replication of the first five levels of rearranged beta matrix **660** and a corresponding interlock table **900**. In various embodiments, interlock tables are used to specify memory access constraints for a sequence of memory accesses. The interlock tables may be stored in an interlock memory system and/or various circuitry may be otherwise configured with the memory access constraints (e.g., rather than maintaining the particular information specified by a given interlock table). The illustrated interlock table(s) are provided to facilitate explanation of particular memory access constraints. In some embodiments, interlock table **900** is generated for a given program or matrix and interlock memory **550** is configured with the memory access constraints specified by interlock table **900**. In the illustrated embodiment, blank entries in interlock table **900** indicate that a corresponding access may be performed at any time because there are no hazards for that access.

[0126] In some embodiments, interlock memory **550** includes a read counter and a write counter, which are initialized to zero before executing a program. In these embodiments, when performing a given read operation, interlock memory **550** may be configured to compare the current write pointer to an entry in interlock table **900** corresponding to the read pointer and allow the read only if the write pointer is greater than the entry. For example, consider the entry in row 3, column 3 in rearranged beta matrix **660**, which indicates location 0. The corresponding entry in interlock table **900** contains a 9. Thus, in this embodiment, a read cannot be performed from location 0 for row 3, column 3 in rearranged beta matrix **660** until the write counter is at least 10. Note that when the write counter is 10, for this sequence of memory operations, location 0 will have been written (based on row 2, column 2 in rearranged beta matrix **660**), and so a potential hazard has been avoided. Similar techniques may be used for writes. For example, in some embodiments, interlock memory **550** is configured to grant a write only if the current read pointer is greater than an entry in the interlock table indicated by the write pointer.

[0127] In some embodiments, interlock memory **550** is not configured to perform hazard checks for write operations because hazards are already resolved by dataflow. For example, for rearranged beta matrix **660**, write after read hazards are not present because the sequence already ensures that reads for a given layer and the previous layer will be completed before a given write to interlock memory **550** is requested (in this embodiment, writes for a given level are not initiated until reads for the entire row have completed in order to find a minimum value).

[0128] FIG. **10** shows a flow diagram illustrating one embodiment of a method **1000** for resolving or preventing data hazards. The method shown in FIG. **10** may be used in conjunction with any of the computer systems, devices, circuits, elements, or components disclosed herein, among others. In various embodiments, some of the method elements (also referred to herein as "blocks") shown may be performed concurrently, in a different order than shown, or may be omitted. Additional method elements may also be performed as desired. Flow begins at block **1010**.

[0129] At block **1010**, arrays for a linearized beta matrix (addr[96]) and interlock table (interlock[96]) are maintained. These arrays may be generated by using raster scan ordering on beta matrix **660** and interlock table **900**. One or both of these arrays are stored in interlock memory **550** in some embodiments. In other embodiments, hazard circuitry is otherwise configured with information from one or both of these arrays. In the illustrated embodiment, rearranged beta matrix **660** may be used as an address table to generate addresses for received memory access requests. Flow proceeds to block **1020**.

[0130] At block **1020**, read and write pointers are initialized to zero. Flow proceeds to blocks **1030** and **1060**.

[0131] At block **1030**, a read request is received. Flow proceeds to decision block **1040**.

[0132] At decision block **1040**, it is determined whether the write pointer is greater than the value in the interlock table at an index of the read pointer (interlock[read pointer]). If the write pointer is greater, flow proceeds to block **1050**. Otherwise, flow proceeds back to decision block **1040** (e.g., inter-

lock memory **550** is configured to wait to grant the read request until the write pointer is greater than the value at interlock[read pointer].

[0133] At block **1050**, the location indicated by a position in the beta matrix at an index of the read pointer (addr[read pointer]) is read. Thus, the requested read operation is performed. Flow returns to block **1030**.

[0134] At block **1060**, a write request is received. Flow proceeds to block **1070**.

[0135] At block **1070**, data for the write request is written to the location indicated by a position in the beta matrix at an index of the write pointer (addr[write pointer]). Thus, the requested write operation is performed. In this embodiment, the write pointer is incremented (which may allow one or more waiting read requests to proceed). Flow returns to block **1060**.

[0136] In other embodiments, writes may be checked in a similar manner as utilized for reads in block **1040**. In these embodiments, interlock memory **550** is configured to stall a given write until the read pointer is greater than interlock [write pointer]. This block is omitted in the illustrated embodiment because of the nature of the writes in the example beta matrix **660**, as discussed above. In still other embodiments, this block may be included for writes but block **1040** may be omitted for reads, e.g., for programs in which dataflow ensures that reads will not cause hazards.

Generating Interlock Information

[0137] FIG. **11** shows a flow diagram illustrating one embodiment of a method **1100** for determining sequences and/or addresses of memory accesses for a program. The method shown in FIG. **11** may be used in conjunction with any of the computer systems, devices, circuits, elements, or components disclosed herein, among others. In various embodiments, some of the method elements (or blocks) shown may be performed concurrently, in a different order than shown, or may be omitted. Additional method elements may also be performed as desired. Flow begins at block **1110**.

[0138] At block **1110**, an input program is received. The program may include memory access operations, e.g., memory reads and/or writes. In some embodiments, a compiler receives the program for compilation for a system that includes interlock memory **550**. Flow proceeds to block **1120**.

[0139] At block **1120**, during compilation of the input program, information is generated by executing portions of the input program. Exemplary techniques for executing portions of an input program during compilation are described in U.S. patent application Ser. No. 13/592,995, filed Aug. 23, 2012 and titled "Compile Time Execution," which is incorporated herein by reference in its entirety. In some embodiments, these techniques are used to generate information for programming interlock memory **550**. Flow proceeds to block **1130**.

[0140] At block **1130**, addresses, a sequence, and/or memory access constraints are generated for memory accesses in the input program based on the information. For example, in some embodiments, a compiler is configured to generate one or more address tables (e.g., a beta matrix) and memory access constraints for one or more interlock tables based on the information. In these embodiments, the compiler may be configured to partially execute the input program in order to determine the order of memory accesses in the program, addresses of the memory accesses, and/or hazards between the memory accesses. In some embodiments, the address information is used for self-addressing. In some embodiments, hazard circuitry for interlock memory **550** is configured with determined memory access constraints and used to avoid hazards when executing the input program. In some embodiments, this configuring may be performed automatically, e.g., when the input program is to be compiled and programmed onto particular hardware. In some embodiments, the compiler is configured to rearrange the sequence of memory operations in order to improve performance using interlock memory **550**. In some embodiments, the rearrangement may alter compilation of the program to use the rearranged sequence. Flow ends at block **1130**.

[0141] In other embodiments, interlock information may be generated using other techniques, in place of and/or in addition to compile time execution. For example, in some embodiments, a program, (e.g., a graphical program) may allow a user to enter an array or matrix corresponding to memory accesses and may be configured to generate interlock and/or address table(s) based on the entered information. For example, in one embodiment, a program is configured to generate a beta matrix and one or more interlock tables based on an input H matrix. In these embodiments, interlock information may be generated before compiling or executing an associated input program at all. In other embodiments any of various programs, arrays, matrices, or other information associated with sets of memory accesses may be input to a program for configuring an interlock memory.

[0142] In various embodiments, information (e.g., interlock information and/or address table(s)) may be transferred to an interlock memory at any of various appropriate times. For example, the information may be transferred at the beginning of execution of a program or while the program is running. In some embodiments, sequence and/or interlock information may be updated during execution to adapt to changing conditions or improve one or more performance measures. Exemplary performance measures include, without limitation: throughput, latency, power consumption, reliability, resource utilization, and bit-error rate.

Performance Estimates

[0143] In some embodiments, a program (e.g., a graphical program) is configured to run cycle accurate simulations for interlock memory **550** for an input program or a sequence of memory access operations. In some embodiments, the cycle accurate simulations are achieved or performed based on knowledge of the sequence of memory operations and the pipelined nature of memory accesses. For example, the number of stalls and number of cycles for each stall may be determined for a particular sequence of accesses. In some embodiments, performance estimates may be generated based on such simulations. In some embodiments, sequences of memory accesses may be reordered to improve performance based on performance estimates.

[0144] FIG. **12** shows a flow diagram illustrating one embodiment of a method for generating a rearranged or reordered sequence of memory access operations based on performance modeling. The method shown in FIG. **12** may be used in conjunction with any of the computer systems, devices, circuits, elements, or components disclosed herein, among others. In various embodiments, some of the method elements shown may be performed concurrently, in a differ-

ent order than shown, or may be omitted. Additional method elements may also be performed as desired. Flow begins at block **1210**.

[0145] At block **1210**, a performance estimate is determined. The performance estimate may indicate an estimated number of cycles needed to perform a sequence of memory access operations using interlock memory **550**. Flow proceeds to decision block **1220**.

[0146] At decision block **1220**, it is determined whether the performance estimate is acceptable. In some embodiments, this determination is based on whether a particular number of iterations of blocks **1210**-**1230** have been performed. In some embodiments, this determination is based on a desired performance estimate or some other threshold. In some embodiments, an optimal ordering may be acceptable, while in other embodiments sub-optimal orderings may be acceptable, e.g., based on some heuristic. If the performance is acceptable, flow proceeds to block **1240** and flow ends. Otherwise, flow proceeds to block **1230**.

[0147] At block **1230**, the sequence of memory access operations is rearranged or reordered. One example of such reordering is discussed above with reference to FIGS. **6**A-**6**C. Flow proceeds to block **1210**. In some embodiments, the rearranged sequences and/or performance estimates are stored, and one of the rearranged sequences is selected based on the performance estimates. In some embodiments, the selected sequence is used to compile a program and/or generate one or more interlock tables for interlock memory **550**.

[0148] FIG. **13** shows a flow diagram illustrating one embodiment of a method for configuring an interlock memory system. The method shown in FIG. **13** may be used in conjunction with any of the computer systems, devices, circuits, elements, or components disclosed herein, among others. In various embodiments, some of the method elements shown may be performed concurrently, in a different order than shown, or may be omitted. Additional method elements may also be performed as desired. Flow begins at block **1310**.

[0149] At block **1310**, a sequence of memory accesses for a program is determined. In some embodiments, the sequence is determined based on compilation of an input program, e.g., per embodiments of the method of FIG. **11**, discussed above. The sequence may be reordered, in some embodiments to improve performance of interlock memory **550**. Flow proceeds to block **1320**.

[0150] At block **1320**, information specifying memory access constraints is generated, based on the sequence of memory accesses. In this embodiment, the information is usable to avoid memory access hazards for the sequence of memory accesses. In some embodiments, the information includes one or more interlock tables. Flow proceeds to block **1330**.

[0151] At block **1330**, circuitry is configured using the information. In this embodiment, the circuitry is included in or coupled to a memory. In this embodiment, the circuitry is operable, after the configuration, to perform memory access requests to the memory corresponding to the sequence of memory accesses while avoiding memory access hazards. In this embodiment, the circuitry is configured to avoid the memory access hazards without receiving other information indicating the memory access hazards (e.g., a program may send the sequence of memory access requests for pipelined execution without indicating memory access hazards and without waiting for a particular memory access to complete before sending dependent accesses). In some embodiment, the configuration is performed using one or more programmable hardware elements. In some embodiments, the configuration is performed by downloading values into one or more storage elements (e.g., one or more random access memories (RAMs) and/or one or more read only memories (ROMs)) included in and/or coupled to interlock memory **550**. In some embodiments, circuitry is also configured with address information for self-addressing the memory accesses using any of various appropriate techniques. In some embodiments, method **1300** is performed automatically based on various inputs to a graphical program. Flow ends at block **1330**.

Additional Interlock Memory Embodiments

[0152] FIG. **14** shows a block diagram illustrating one embodiment of a system **1400** configured to implement an interlock memory. FIGS. **15**A-B and **16**A-B illustrate exemplary interlock tables, execution timelines, and methods associated with system **1400** in some embodiments. System **1400**, in the illustrated embodiment, includes memory **1410** and hazard circuitry **1420**. Memory **1410**, in some embodiments, is a shared memory.

[0153] Hazard circuitry **1420**, in the illustrated embodiment, is configured to receive read and write requests and determine when to grant them in order to resolve hazards based on write interlock table **1470** and read interlock table **1480**. Hazard circuitry **1420**, in some embodiments, is integrated into the memory system that includes memory **1410**. Hazard circuitry **1420**, in the illustrated embodiment, includes write interlock table **1470**, read interlock table **1480**, and circuitry configured to maintain values for write pointer **1430**, read pointer **1440**, write tokens **1450**, and read tokens **1460**. In various embodiments, hazard circuitry may be distributed among elements of a pipelined data path. In some embodiments the information specified by write interlock table **1470** and read interlock table **1480** may not be directly stored, but may instead be used to configure hazard circuitry to perform various functionality described below.

[0154] Write tokens **1450** and read tokens **1460**, in the illustrated embodiment, indicate how many reads or writes can be respectively performed before the next hazard, assuming the tokens are not incremented. For example, if read tokens **1460** has a value of three, then three reads can be performed before a potential hazard, assuming the value is not incremented as a result of writes in the interim.

[0155] Write pointer **1430** and read pointer **1440**, in the illustrated embodiment, indicate the how many operations of each type have been performed in a sequence of memory accesses. In some embodiments, write pointer **1430** and read pointer **1440** are initialized to zero before execution of a given program and are incremented for each write and read respectively.

[0156] Write interlock table **1470** and read interlock table **1480**, in the illustrated embodiment, indicate values to be added to read tokens **1460** or write tokens **1450** upon completion of particular read or write operations. These tables may be generated using compile time execution, generated based on user input, hard coded for a particular array or matrix, etc. Exemplary embodiments of values for these tables are discussed below with reference to FIGS. **15**A-B.

[0157] Hazard circuitry **1420**, in the illustrated embodiment, is configured to provide write address, write enable, read address, and read enable signals to memory **1410**. In some embodiments, hazard circuitry **1420** is self-addressing

and does not receive the address information during execution of a sequence of memory operations. In other embodiments, the addresses are included in the read and write requests.

[0158] FIGS. **15**A-B show exemplary embodiments of interlock tables and pipelined execution for different exemplary sequences of memory access operations using system **1400**. These Figures show that rearranging a sequence of memory accesses (e.g., rearranging beta matrix **1510** to produce beta matrix **1550** in the illustrated embodiment) may increase pipelined performance memory accesses using an interlock memory.

[0159] Beta matrix **1510**, in the illustrated embodiment, includes two layers that each include three blocks. In the illustrated embodiment, the positions in the second layer of each table are shown using shaded blocks. Beta matrix **1550**, in the illustrated embodiment, is generated by reordering the first layer of beta matrix **1510**.

[0160] The execution diagrams illustrate exemplary pipelined performance of memory accesses for each of beta matrices **1510** and **1550**. In the illustrated embodiment, upon completion of memory accesses for layer 2, execution begins again for layer 1. FIGS. **15**A-B show that rearranging the order of memory access operations can improve performance, as the reordering to generate beta matrix **1550** has improved throughput relative to beta matrix **1510**.

[0161] Write interlock tables **1520** and **1560** and read interlock tables **1530** and **1570**, in the illustrated embodiment, include entries indicating the number of tokens to add upon completion of a corresponding operation in one of the beta matrices. For example, upon completion of a write to location 0 for layer 1 of beta matrix **1510**, two read tokens are added, allowing at least two reads to occur before a stall to avoid the next potential hazard. As another example, upon completion of a write to location 4 for layer 2 of beta matrix **1510**, four read tokens are added, allowing at least for reads to occur before a stall to avoid the next hazard. In the illustrated embodiment, each read operation adds a single write token. Note that in the illustrated embodiments, writes do not occur immediately after a token is added. In this embodiment, this is the result of the nature of the LDPC decoding operations, in which a program is configured to read all positions in a layer before writing any of them. Thus, when the first write request is received, in this example, the write tokens value has already been incremented to three based on the three reads.

[0162] FIGS. **16**A-B show exemplary embodiments of methods **1600** and **1650** for operation of an interlock memory system, such as, for example, the embodiment of system **1400**. The methods shown in FIGS. **16**A-B may be used in conjunction with any of the computer systems, devices, circuits, elements, or components disclosed herein, among others. In various embodiments, some of the method elements shown may be performed concurrently, in a different order than shown, or may be omitted. Additional method elements may also be performed as desired. Flow begins at block **1602** for FIG. **16**A and block **1652** for FIG. **16**B. In some embodiments, these methods are performed (at least partially) in parallel.

[0163] Referring to FIG. **16**A, at block **1602**, hazard circuitry initializes a read tokens value. In the embodiments of FIGS. **15**A-B, read tokens are initialized to a value of three while write tokens are initialized to a value of zero. Flow proceeds to block **1604**.

[0164] At block **1604**, hazard circuitry **1420** waits for a read request. Flow proceeds to decision block **1606**.

[0165] At decision block **1606**, hazard circuitry **1420** determines whether read tokens is greater than zero. If it is, flow proceeds to block **1608**. Otherwise, flow remains at decision block **1606** until read tokens is greater than zero. As shown by the dashed arrow from block **1666** in FIG. **16**B, read tokens may become greater than zero in response to a write operation, e.g., because performance of the write operation removes a hazard.

[0166] At block **1608** hazard circuitry **1420** fetches a token value from read interlock table **1470** based on the read pointer. In self-addressing embodiments, hazard circuitry **1420** is also configured to retrieve the read address from an address table. Note that in some embodiments, the interlock table may be configured to store addresses as well as indicate memory constraints. In these embodiments addresses may be retrieved from the interlock table in addition to interlock information. Flow proceeds to block **1610**.

[0167] At block **1610**, hazard circuitry **1420** increments the read pointer. Flow proceeds to block **1612**.

[0168] At block **1612**, hazard circuitry **1420** decrements the read tokens. In this embodiment, each read may be thought of as consuming a read token, such that reads are allowed to occur only if a read token is available. Flow proceeds to block **1614**.

[0169] At block **1614**, hazard circuitry **1420** performs the requested read operation. Flow proceeds to block **1616**.

[0170] At block **1616**, hazard circuitry **1420** adds the retrieved token value from block **1608** to write tokens. As shown by the dashed line from block **1616** to decision block **1656** of FIG. **16**B, this incrementing may allow a waiting write to proceed because performance of the read in block **1614** may remove a hazard. Flow returns to block **1604**.

[0171] Referring to FIG. **16**B, at block **1652**, hazard circuitry **1420** initializes a write tokens value. Flow proceeds to block **1654**.

[0172] At block **1654**, hazard circuitry **1420** waits for a write request. Flow proceeds to decision block **1656**.

[0173] At decision block **1656**, hazard circuitry **1420** determines whether write tokens is greater than zero. If it is, flow proceeds to block **1658**. Otherwise, flow remains at decision block **1656** until write tokens is greater than zero.

[0174] At block **1658** hazard circuitry **1420** fetches a token value from write interlock table **1480** based on the write pointer. In self-addressing embodiments, hazard circuitry **1420** is also configured to retrieve the write address from an address table. Flow proceeds to block **1660**.

[0175] At block **1660**, hazard circuitry **1420** increments the write pointer. Flow proceeds to block **1662**.

[0176] At block **1662**, hazard circuitry **1420** decrements the write tokens. In this embodiment, each write may be thought of as consuming a write token, such that writes are allowed to occur only if a write token is available. Flow proceeds to block **1664**.

[0177] At block **1664**, hazard circuitry **1420** performs the requested write operation. Flow proceeds to block **1666**.

[0178] At block **1666**, hazard circuitry **1420** adds the retrieved token value from block **1608** to read tokens. As shown by the dashed line from block **1666** to decision block **1606** of FIG. **16**A, this incrementing may allow a waiting read to proceed because performance of the write in block **1654** may remove a hazard. Flow returns to block **1654**.

[0179] Note that FIGS. **14-16**B illustrate exemplary token-based techniques for interlock tables while FIG. **9** illustrates an exemplary counter-based technique (e.g., each entry in interlock table **900** indicates the count or number of memory operations that must occur before a given access is allowed to proceed). In other embodiments, various other techniques may be used to indicate hazards using an interlock table and stall execution of memory accesses based on the interlock table. The disclosed embodiments are provided herein for exemplary purposes and are not intended to limit interlock table implementations in other embodiments.

[0180] In some embodiments, an interlock memory system may be configured with sets of memory access constraints for multiple different programs. In these embodiments, an indication of one of the different programs may be transmitted to the interlock memory system and the corresponding memory access constraints may be used to handle memory access requests for a given program. In various programmable embodiments, circuitry configured with a first set of memory access constraints for a first program may be re-configured with a second, different set of memory access constraints for a second program.

[0181] Re-configurable circuitry may be advantageous in various applications. For example, a user equipment device (UE) that includes re-configurable interlock circuitry may receive a new H matrix or beta matrix for LDPC-backed communications. In these embodiments, the UE may be configured to generate a program to implement LDPC decoding based on the received matrix. In other embodiments, the program may be transferred directly to the UE. In these embodiments, the UE may be configured to re-order memory accesses corresponding to the matrix in order to reduce pipelined execution time. In these embodiments, the UE is configured to configure hazard circuitry with a new set of memory access constraints for the communication. In some embodiments, the UE is configured to receive a set of memory access constraints for the program rather than generating them internally. In various embodiments, the UE may decode incoming wireless transmissions using the program and an interlock memory configured with the memory access constraints. In some embodiments, a UE that includes interlock memory configured to store memory access constraints for multiple programs may be configured to use different parity check techniques for different communications, e.g., when a device communicates via different networks.

[0182] Referring now to FIG. **17**, one generalized embodiment of a method **1700** for operating an interlock memory is shown. The method shown in FIG. **17** may be used in conjunction with any of the computer systems, devices, circuits, elements, or components disclosed herein, among others. In various embodiments, some of the method elements shown may be performed concurrently, in a different order than shown, or may be omitted. Additional method elements may also be performed as desired. Flow begins at block **1710**.

[0183] At block **1710**, a sequence of memory access requests is received for a memory. In some embodiments, execution of a program generates the sequence of memory access requests. Flow proceeds to block **1720**.

[0184] At block **1720**, the memory access requests are granted by circuitry configured with memory access constraints for the memory access requests. In this embodiment, the circuitry is configured to grant the memory access requests subject to the memory access constraints, thereby avoiding memory access hazards for a sequence of memory accesses corresponding to the sequence of memory access requests. The circuitry may include one or more storage elements configured to store information specifying the memory access constraints. The circuitry may be programmable to grant the memory access requests based on the memory access constraints (e.g., the circuitry may include programmable hardware elements). The circuitry may be dedicated to granting the memory access requests based on the memory access constraints (e.g., the circuitry may be an ASIC). Flow ends at block **1720**.

[0185] Referring now to FIG. **18**, one generalized embodiment of a method **1800** for reordering a sequence of memory accesses is shown. The method shown in FIG. **18** may be used in conjunction with any of the computer systems, devices, circuits, elements, or components disclosed herein, among others. In various embodiments, some of the method elements shown may be performed concurrently, in a different order than shown, or may be omitted. Additional method elements may also be performed as desired. Flow begins at block **1810**.

[0186] At block **1810**, a specified sequence of memory accesses is stored. The specified sequence may be received from another computing system or generated by the computing system storing the sequence, e.g., using compile-time execution. In this embodiment, the specified sequence corresponds to a function to be performed (such as LDPC decoding, for example) and has first memory access constraints. Flow proceeds to block **1820**.

[0187] At block **1820**, the specified sequence is reordered to create a reordered sequence of memory accesses that has second, different memory access constraints. In this embodiments, the reordered sequence of memory accesses is usable to access a memory to perform the function. The reordering may improve performance of the memory accesses using an interlock memory, as discussed above with reference to FIGS. **6**C and **12**. The reordering may be performed multiple times and a particular reordered sequence may be selected based on one or more criteria. The reordered sequence of memory accesses may have a greater memory access efficiency than the original sequence of memory accesses. For example, the reordered sequence may be usable to perform the function in a smaller number of clock cycles than a number of clock cycles used to perform the function using the original sequence. Flow ends at block **1820**.

[0188] In some embodiments, a computing system may compile a program to implement the reordered sequence of memory accesses. Execution of the program may generate a sequence of memory access requests corresponding to the reordered sequence of memory accesses. The memory access requests may be performed using an interlock memory.

[0189] Referring now to FIG. **19**, one generalized embodiment of a method **1900** for operating a self-addressing memory is shown. The method shown in FIG. **19** may be used in conjunction with any of the computer systems, devices, circuits, elements, or components disclosed herein, among others. In various embodiments, some of the method elements shown may be performed concurrently, in a different order than shown, or may be omitted. Additional method elements may also be performed as desired. Flow begins at block **1910**.

[0190] At block **1910**, memory access requests are received, corresponding to a specified sequence of memory accesses, but the memory access requests do not include address information. For example, the memory access requests may be generated by an LDPC decoder program.

The requests may specify whether a read or write operation is desired, but do not specify a location to be read/written. Flow proceeds to block **1920**.

[0191] At block **1920**, addresses are assigned to the memory accesses requests for the specified sequence of memory accesses. This may be performed by addressing circuitry configured with address information for the specified sequence of memory accesses. The addressing circuitry may be programmable or may be dedicated to a particular sequence. Flow proceeds to block **1930**.

[0192] At block **1930**, the sequence of memory accesses is performed using the assigned addresses. Flow ends at block **1930**.

[0193] Referring now to FIG. **20**, a sequence of memory accesses **2020** for Cholesky factorization of a symmetric matrix **2010** is shown. In the illustrated embodiment, symmetric matrix **2010** is divided into 2×2 submatrix blocks B**0**-B**3**. Sequence **2020** captures the Choleskey factorization algorithm's sequence of read/modify/writes for these submatrix blocks. In the illustrated embodiment, for the second row, for example, B**2** and B**0** are read, potentially modified, and written before proceeding to row three. In other embodiments, similar sequences may be determined for any of various matrix operations. In some embodiments, various techniques described herein may be utilized for such sequences. For example, interlock information may be generated for sequence **2020**, sequence **2020** may be rearranged to reduce memory access hazards, addresses may be generated for sequence **2020**, etc.

[0194] Although the embodiments above have been described in considerable detail, numerous variations and modifications will become apparent to those skilled in the art once the above disclosure is fully appreciated. It is intended that the following claims be interpreted to embrace all such variations and modifications.

We claim:

1. A method, comprising:

storing, by a computing system, a specified sequence of memory accesses, wherein the specified sequence corresponds to a function to be performed, wherein the specified sequence of memory accesses has first memory access constraints; and

reordering, by the computing system, the specified sequence of memory accesses to create a reordered sequence of memory accesses, wherein the reordered sequence of memory accesses has second memory access constraints, wherein the second memory access constraints are different than the first memory access constraints;

wherein the reordered sequence of memory accesses is usable to access a memory to perform the function.

2. The method of claim **1**, further comprising:

determining performance estimates for a plurality of reordered sequences of memory accesses, wherein the plurality of reordered sequences of memory accesses are based on the specified sequence of memory accesses and are usable to access the memory to perform the function; and

selecting one of the plurality of reordered sequences based on the performance estimates.

3. The method of claim **2**,

wherein the performance estimates are based on memory access constraints for each of the plurality of reordered sequences of memory accesses; and

wherein the performance estimates are determined based on cycle-accurate simulations of the reordered sequences of memory accesses for pipelined performance of the memory accesses using hazard circuitry.

4. The method of claim **1**, further comprising:

configuring first circuitry coupled to or comprised in a memory according to the second memory access constraints, wherein said configuring configures the first circuitry to implement the second memory access constraints.

5. The method of claim **4**, further comprising:

executing a program to generate a sequence of memory access requests corresponding to the reordered sequence of memory accesses; and

granting, by the first circuitry, the sequence of memory access requests according to the second memory access constraints, thereby avoiding memory hazards for the sequence of memory access requests.

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

compiling a program to implement the reordered sequence of memory accesses.

7. The method of claim **1**,

wherein the specified sequence of memory accesses has a first memory access efficiency; and

wherein the reordered sequence of memory accesses has a second, greater memory access efficiency.

8. The method of claim **1**, wherein the specified sequence of memory accesses is based on a parity check matrix.

9. A system, comprising:

one or more processors; and

one or more memories having program instructions stored thereon that are executable by the one or more processors to perform operations comprising:

storing a specified sequence of memory accesses, wherein the specified sequence corresponds to a function to be performed, wherein the specified sequence of memory accesses has first memory access constraints; and

reordering the specified sequence of memory accesses to create a reordered sequence of memory accesses, wherein the reordered sequence of memory accesses has second memory access constraints, wherein the second memory access constraints are different than the first memory access constraints;

wherein the reordered sequence of memory accesses is usable to access the memory to perform the function.

10. The system of claim **9**, wherein the operations further comprise:

determining performance estimates for a plurality of reordered sequences of memory accesses, wherein the plurality of reordered sequences of memory accesses are based on the specified sequence of memory accesses and are usable to access the memory to perform the function; and

selecting one of the plurality of reordered sequences based on the performance estimates.

11. The system of claim **9**, wherein the operations further comprise:

configuring first circuitry coupled to or comprised in a memory according to the second memory access constraints, wherein said configuring configures the first circuitry to implement the second memory access constraints for the reordered sequence of memory accesses.

12. The system of claim **11**, wherein the operations further comprise:

generating a sequence of memory access requests corresponding to the reordered sequence of memory accesses; and

granting, by the first circuitry, the sequence of memory access requests according to the second memory access constraints, thereby avoiding memory hazards for the sequence of memory access requests.

13. The system of claim **9**, wherein the operations further comprise:

compiling a program to implement the reordered sequence of memory accesses.

14. The system of claim **9**, wherein the operations further comprise:

determining the specified sequence of memory accesses using compile-time execution of an input program.

15. The system of claim **9**,

wherein the specified sequence of memory accesses has a first memory access efficiency; and

wherein the reordered sequence of memory accesses has a second, greater memory access efficiency.

16. The system of claim **9**, wherein the specified sequence of memory accesses corresponds to entries in a beta matrix, wherein the beta matrix is generated based on a parity check matrix.

17. A non-transitory computer-readable medium having instructions stored thereon that are executable by a computing device to perform operations comprising:

storing a specified sequence of memory accesses, wherein the specified sequence corresponds to a function to be performed, wherein the specified sequence of memory accesses has first memory access constraints; and

reordering the specified sequence of memory accesses to create a reordered sequence of memory accesses, wherein the reordered sequence of memory accesses has second memory access constraints, wherein the second memory access constraints are different than the first memory access constraints;

wherein the reordered sequence of memory accesses is usable to access the memory to perform the function.

18. The non-transitory computer-readable medium of claim **17**, wherein the operations further comprise:

reordering the specified sequence of memory accesses to create one or more additional, different reordered sequences of memory accesses;

determining performance estimates for the reordered sequence of memory accesses and the one or more additional, different reordered sequences of memory accesses, wherein the performance estimates are based on memory access constraints for the reordered sequence and the one or more different, reordered sequences; and

selecting one of the reordered sequence and the one or more additional, different reordered sequences based on the performance estimates.

19. The non-transitory computer-readable medium of claim **17**, wherein the operations further comprise:

configuring first circuitry coupled to or comprised in a memory according to the second memory access constraints, wherein said configuring configures the first circuitry to implement the second memory access constraints for the reordered sequence of memory accesses.

20. The non-transitory computer-readable medium of claim **17**, wherein the operations further comprise:

determining the specified sequence of memory accesses using compile-time execution of an input program.

* * * * *